Real-World Cryptography

David Wong

MEAP

MANNING

MEAP Edition
Manning Early Access Program
Real-World Cryptography
Version 12

For more information on this and other Manning titles go to
manning.com

welcome

Dear reader,

Getting into cryptography is a road paved with difficult challenges and many uncertainties about what there is to learn, or what's important to know. "What's the quickest path to get there?" you may have often asked yourself. And you were right to wonder. The amount of information out there can be intimidating, especially when most of it is outdated or goes deeply into theoretical rabbit holes.

Fear not, you've come to the right place. *Real-World Cryptography* is here for you, the curious, the student, the engineer who wants to know (or needs to know) more about cryptography.

In this book, I first summarize the state of real-world cryptography, which is the cryptography that is used every day by you and me, as well as the large companies of this world. Almost everything you learn in this book is practical and useful.

The rest of the book looks into what the real-world cryptography of tomorrow is starting to look like. Many topics like cryptocurrencies and post-quantum cryptography are often ignored from the applied cryptography literature, but they are equally as important. The field of applied cryptography is currently booming and changing rapidly, and one has to keep up with its advances or fear being left out behind.

This is not a reference book, and so I have avoided most math (although not all) and deeply technical details. Most of what you will learn will first teach you about cryptographic objects as if they were black boxes that serve useful purposes if given the right inputs. The book also strives to give you the right amount of detail, just enough to push your curiosity forward and give you a peek of what these black boxes look like from the inside. Each chapter is accompanied with many real-world examples of what you just learned.

There are no specific prerequisites for reading this book aside from having a taste for computer science and having heard the word "encryption" before. Students and engineers should be able to pick up this book and have most of their questions answered. Please feel free to join in on the conversation in the book's liveBook Discussion Forum. Your questions and comments are welcome and appreciated, and will help make this book the best it can be.

Prepare yourself, as I will now demystify the real world of cryptography before your eyes.

-David Wong

brief contents

1 Introduction

This chapter covers

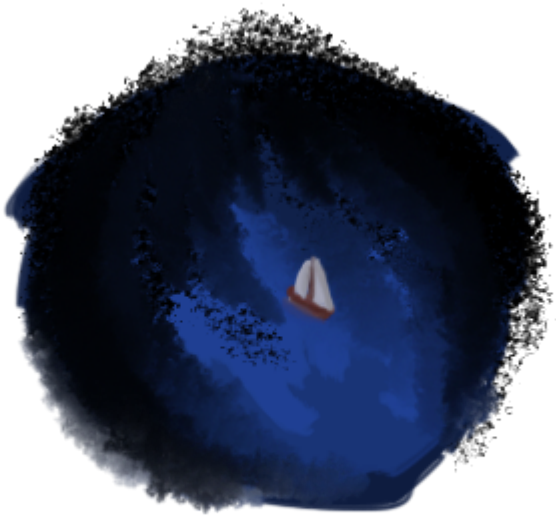
- What cryptography is about.
- The difference between theoretical cryptography and real-world cryptography.
- What you will learn throughout this adventure.

Greetings traveler,

Sit tight as you're about to enter a world of wonder and mystery—the world of cryptography. Cryptography is the ancient discipline of securing situations that are troubled with malicious characters. In this book are the spells we need to defend ourselves against the malice. Many have attempted to learn this craft, but few have survived the challenges that stand in the way of mastery. Exciting adventures await! We'll uncover how cryptographic algorithms can secure our letters, identify our allies, and protect treasures from our enemies. Sailing through the cryptographic sea will not be the smoothest journey, as cryptography is the foundation of all security in our world, the slightest mistake could be deadly.

SIDEBAR **Remember**

If you find yourself lost, keep moving forward. It will all eventually make sense.



1.1 Cryptography is about securing protocols

Our journey starts here with an introduction to cryptography, the science aiming to defend protocols against saboteurs. But first, what's a protocol? Simply put: it's a list of steps that one (or more people) must follow in order to achieve something. For example, imagine the following premise: you want to leave your magic sword unattended for a few hours so you can take a nap. One protocol to do this could be the following:

1. Deposit weapon on the ground.
2. Take nap under a tree.
3. Recover weapon from the ground.

Of course, it's not a great protocol as anybody can steal your sword while you're napping… And so cryptography is about taking into account these adversaries who are looking to take advantage of you.

In ancient times, when rulers and generals were busy betraying each other and planning coups, one of their biggest problems was finding a way to **share confidential information with those they trusted**. From here the idea of cryptography was born. It took centuries and hard work before cryptography became the serious discipline it is today. Now, it's used all around us, to provide the most basic services in the face of our chaotic and adverse world.

The story of this book is about the practice of cryptography on our planet. It takes you on an

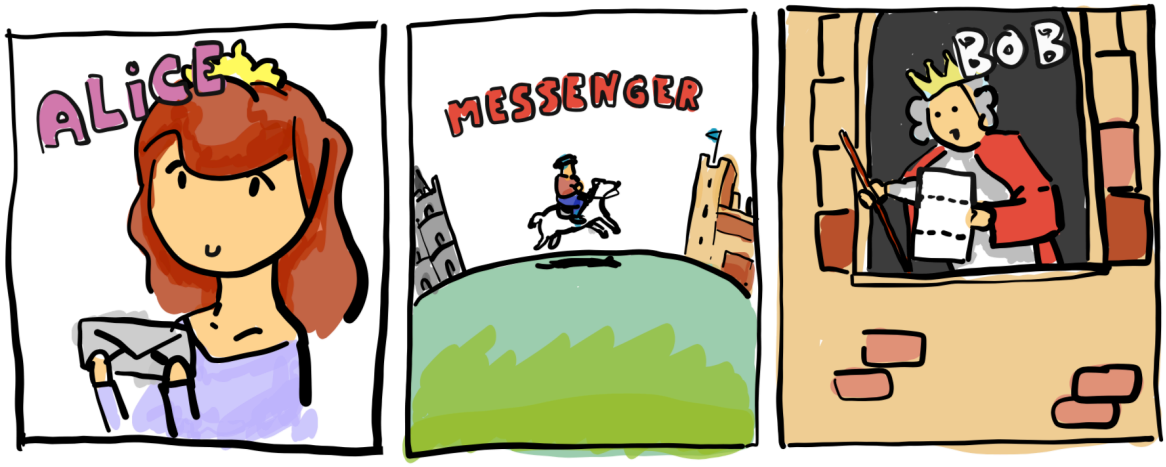
expedition throughout the computing world to cover cryptographic protocols in use today, it shows you what are the parts they are made of, and how everything fits together. While your typical cryptography book usually starts with the discovery of cryptography, and takes you through its history, I think that it makes little sense for me to kick off things this way. I want to tell you about the practical. I want to tell you about what I've witnessed myself, reviewing cryptographic applications for large companies as a consultant, or the cryptography I've made use of myself as an engineer in the field.

There will be (almost) no scary math formulas. The purpose of this book is to demystify cryptography, survey what is considered useful nowadays, and give intuition about how things around you are built. This book is intended for the curious people, the interested engineers, the adventurous developers and the inquisitive researchers.

Chapter 1 initiates a tour of the world of cryptography. We will discover the different types of cryptography, which ones matter to us, and how the world agreed on using them.

1.2 Symmetric cryptography: what is symmetric encryption?

One of the fundamental concepts of cryptography is **symmetric encryption**. It is used in a majority of cryptographic algorithms in this book, and it is thus extremely important. We introduce this new concept here via our very first protocol. Let's imagine that **queen Alice** needs to send a letter to **lord Bob** who is a few castles away. She asks her loyal messenger to ride his camel and battle his way through the filthy lands ahead in order to deliver the precious message to lord Bob. Yet, she is suspicious; even though her loyal messenger has served her for many years, she wishes the message in transit to **remain secret from all passive observers, including the messenger**. You see, the letter most likely contains some controversial gossip about the kingdoms on the way.



What queen Alice needs is a protocol that acts like she would effectively hand the message to lord Bob herself. No middle men. This is quite an impossible problem to solve in practice, unless

we introduce cryptography (or teleportation) into the equation. And this is what we ended up doing ages ago by inventing a new type of cryptographic algorithm called a **symmetric encryption algorithm** (also known as a **cipher**).

> **NOTE** By the way, a type of cryptographic algorithm is often referred to as a primitive. You can think of a primitive as the smallest useful construction you can have in cryptography, and it is often used with other primitives in order to build a protocol. It is mostly a term, and has no particularly important meaning, though it appears often enough in the literature that it is good to know about it.

Let's see next how an encryption primitive can be used to hide queen Alice's message from the messenger. Imagine for now that the primitive is a black box (we can't see what it's doing internally) that provides two functions:

- `ENCRYPT`
- `DECRYPT`

The first function, **ENCRYPT** works by taking a **secret key** (usually a very large number) and a **message**. It then outputs a series of numbers that look like they were chosen randomly, some **noisy** data if you will. We will call that output the encrypted message. I illustrate this in figure 1.1.

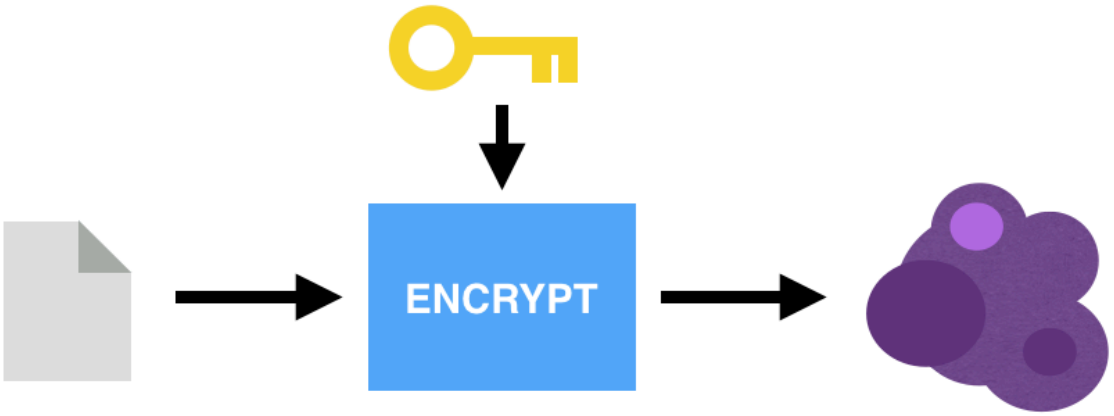


Figure 1.1 The ENCRYPT function takes a message and a secret key, and outputs the encrypted message—a long series of numbers that look like random noise.

The second function, **DECRYPT**, is the inverse of the first one: by taking the same **secret key** and the random output of the first function (the encrypted message), it finds back the original message.

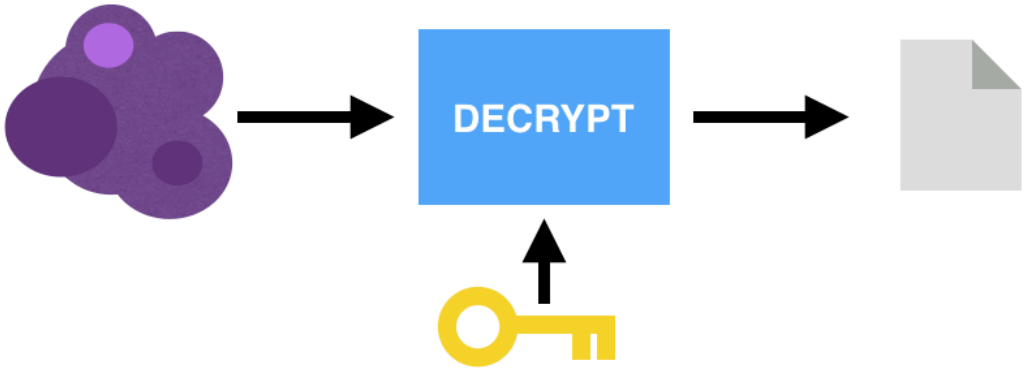


Figure 1.2 The DECRYPT function takes an encrypted message and a secret key, and returns the original message.

To make use of this new primitive, queen Alice and lord Bob have to first meet in real life and decide on what **secret key** to use. Later, queen Alice can use the provided **ENCRYPT** function to protect a message with the help of the **secret key**. She then passes the encrypted message to her messenger, who eventually delivers it to lord Bob. Lord Bob then uses the **DECRYPT** function on the encrypted message, with the same secret key, to recover the original message.

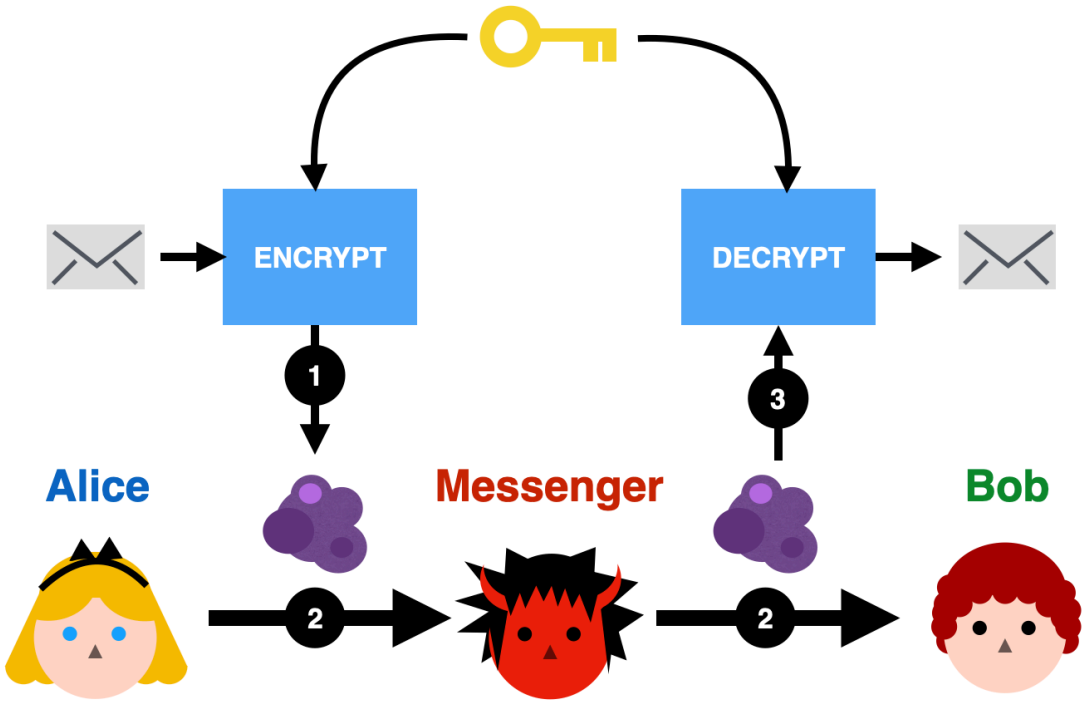


Figure 1.3 (1) Alice uses the ENCRYPT function with a secret key to transform her message into noise. (2) She then passes the encrypted message to her messenger, who will not learn anything about the underlying message. (3) Once Bob receives the encrypted message, he can recover the original content by using the DECRYPT function with the same secret key Alice used.

During this exchange, all the messenger had was something that looked random and that

provided no meaningful insight into the content of the hidden message. Effectively, we augmented our insecure protocol into a secure one thanks to the help of cryptography. The new protocol makes it possible for queen Alice to deliver a confidential letter to lord Bob without anyone (but lord Bob) learning the content of it. The process of using a secret key to render things to noise, making them **indistinguishable from random**, is a common way of securing a protocol in cryptography. We will see more of this as we learn more cryptographic algorithms in the next chapters.

By the way, symmetric encryption is part of a larger category of cryptography algorithms called **symmetric cryptography** or **secret-key cryptography**. This is due to the **same key** being used by the different functions exposed by the cryptographic primitive. As you will see later, sometimes there are more than one key.

1.3 Kerckhoff's principle: only the key is kept secret

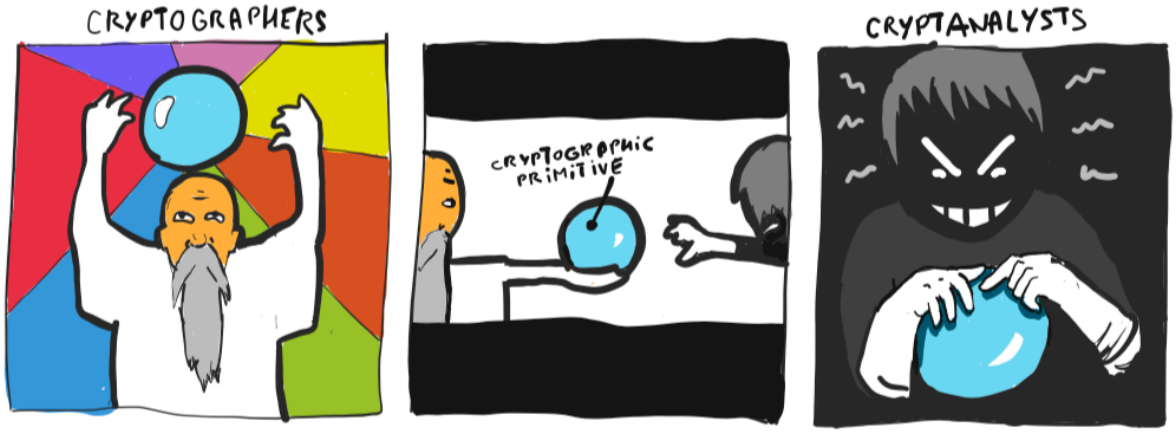
To design a cryptographic algorithm (like our encryption primitive) is an easy task, but to design a **secure** cryptographic algorithm is not for the faint of heart. While we shy away from creating such algorithms in this book, we *do* learn how to recognize the good ones to use. This can be difficult, as there is more choice than one can ask for the task. Hints can be found in the repeated failures of the history of cryptography, as well as the lessons that the community has learned from them. As we take a look at the past, we will grasp at what turns a cryptographic algorithm into a trusted-to-be-secure one.

Hundreds of years have passed and many queens and lords have been buried. Since then, paper has been abandoned as our primary means of communication in favor of better and more practical technologies. Today we have access to powerful computers as well as the internet. More practical, sure, but this also means that our previous malicious messenger has become much more powerful. He is now everywhere: the Wi-Fi in the Starbucks cafe you're sitting in, the different servers making up the internet and forwarding your messages, the machines running our algorithms. Our enemies are now able to observe many more messages, as each request you make to a website might pass through the wrong wire, and become altered or copied in a matter of nanoseconds without anyone noticing.

Before us, we can see that old and recent history contains many instances of encryption algorithms falling apart, being broken by secret state organizations or by independent researchers, and failing to protect their messages or accomplish their claims. Many lessons were learned, and we slowly came to understand how to produce good cryptography.

> **NOTE** A cryptographic algorithm can be considered broken in many ways. For an encryption algorithm, you can imagine several ways to attack the algorithm: the secret key can be leaked to the attackerm messages can be decrypted without the help of the key, some information about the message can be revealed just by looking at the encrypted message, and so on. Anything that would somehow weaken the assumptions we made about the algorithm could be considered a break.

A strong notion came out of the long process of trial and errors that cryptography went through: to obtain confidence in the security claims made by a cryptographic primitive, the primitive has to be analyzed in the open by experts. Short of that, you are relying on **security through obscurity**, which hasn't worked well historically. This is why **cryptographers** (the people who build) will usually use the help of **cryptanalysts** (the people who break) in order to analyze the security of a construction. (Although cryptographers are often cryptanalysts themselves, and vice-versa.)



Let's take the **Advanced Encryption Standard (AES)** encryption algorithm as an example. AES was the product of an international competition organized by the National Institute of Standards and Technology (NIST).

> **NOTE** The NIST is a United States agency whose role is to define standards and develop guidelines for use in government-related functions as well as other public or private organizations. It has standardized many widely-used cryptographic primitives like AES.

The AES competition lasted several years, during which many volunteering cryptanalysts from around the world gathered to take a chance at breaking the various candidate constructions. After several years, once enough confidence was built by the process, a single competing encryption algorithm was nominated to become the Advanced Encryption Standard itself. Nowadays, most

people trust that AES to be a solid encryption algorithm, and it is widely used to encrypt almost anything. For example, you use it every day when you browse the web.

The idea to build cryptographic standards in the open is related to a concept often referred to as **Kerckhoffs' principle**, which can be understood as: it would be foolish to rely on our enemies not to discover what algorithms we use, because they most likely will. Instead, let's be open about them.

If the enemies of queen Alice and lord Bob knew exactly how they were encrypting messages, how was our encryption algorithm secure? The **secret key**! The secrecy of the key makes the protocol secure, not the secrecy of the algorithm itself. This is a common theme in this book: all the cryptographic algorithms that we will learn about, and that are used in the real-world, are most often free to be studied and used. Only the secret keys that are used as input to these algorithms, when being used, are kept secret. "*Ars ipsi secreta magistro*" (an art secret even to its inventor) said Jean Robert du Carlet in 1644.

In the next section, we will talk about a totally different kind of cryptographic primitive, so let's use figure 1.4 to organize what we've learned so far.

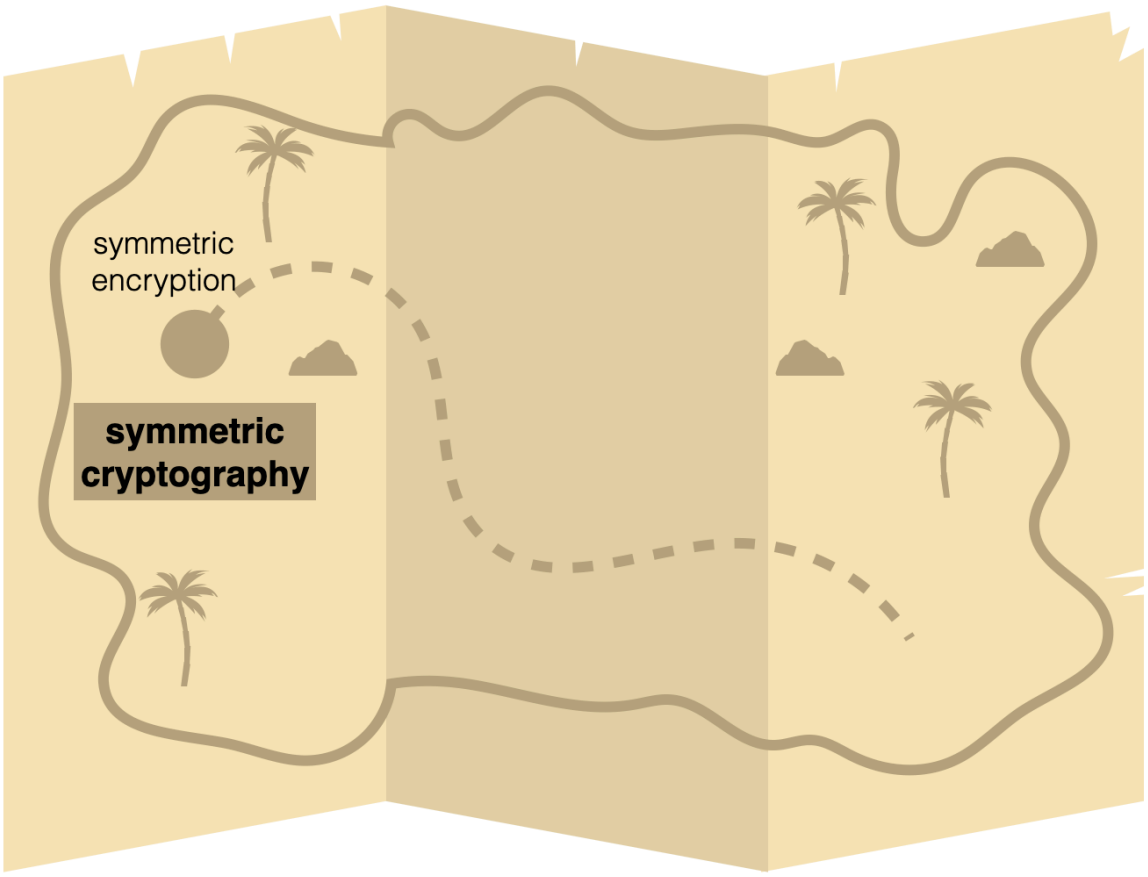


Figure 1.4 The cryptographic algorithms we have learned so far. AES is an instantiation of a symmetric encryption algorithm, which is a cryptographic primitive that is part of the broader class of symmetric cryptographic algorithms. There are many more of those (represented as question marks) that we will learn about in this book.

1.4 Asymmetric cryptography: two keys are better than one

In our discussion about symmetric encryption, we said that Alice and Bob first met to decide on a symmetric key. This is a plausible scenario, and a lot of protocols actually do work like this. Nonetheless, this quickly becomes less practical in protocols with many participants: do we need our web browser to meet with Google, Facebook, Amazon, and the other billions of websites before being able to securely connect to them? This problem, often referred to as **key distribution** has been a hard one to solve for quite a long time, at least until the discovery in the late 70s of another large and useful category of cryptographic algorithms called **asymmetric cryptography** or **public-key cryptography**. Asymmetric cryptography generally makes use of different keys for different functions (as opposed to a single key being used in symmetric cryptography), or provides different point of views to different participants. To illustrate what this mean, and how public-key cryptography helps to set up trust between people, I'll introduce a number of asymmetric primitives in this section.

Note that this is only a glance of what you'll learn in this book, as I'll talk about each of these

cryptographic primitives in more detail in subsequent chapters.

1.4.1 Key exchanges, or how to get a shared secret

The first asymmetric cryptography primitive we'll look at is the key exchange one.

The first public-key algorithm discovered and published was a key exchange algorithm named after its authors—**Diffie-Hellman**. The Diffie-Hellman key exchange algorithm's main purpose is to establish a common secret between two parties. This common secret can then be used for different purposes, for example, as a key to a symmetric encryption primitive.

In chapter 6, I will explain in detail how Diffie-Hellman works, but for this introduction let's use a simple analogy in order to understand what a key exchange provides. Like many algorithms in cryptography, a key exchange must start with the participants using a common set of parameters. In our analogy, we will simply have them agree to use a square ■.

The next step is for Alice and Bob to choose their own random shape. Both of them go to their respective secret place, and out of sight, Alice chooses a triangle ▲ and Bob chooses a star . The objects they have chosen need to remain secret at all costs! The objects represents their **private keys** (see figure 1.5).

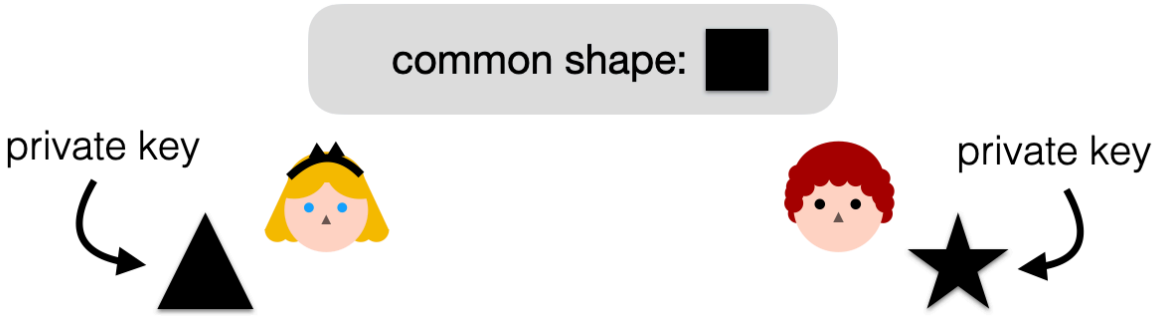


Figure 1.5 The first step of a Diffie-Hellman key exchange is to have both participants generate a private key. In our analogy, Alice chooses a triangle as her private key, whereas Bob chooses a star as his private key.

Once they have done that, they both individually combine their secret shape with the common shape they initially agreed on using. This gives out a unique shape, which represents their **public key**. Alice and Bob can now exchange their public keys (hence the name "key exchange") as public keys are considered public information. I illustrate this in figure 1.6.

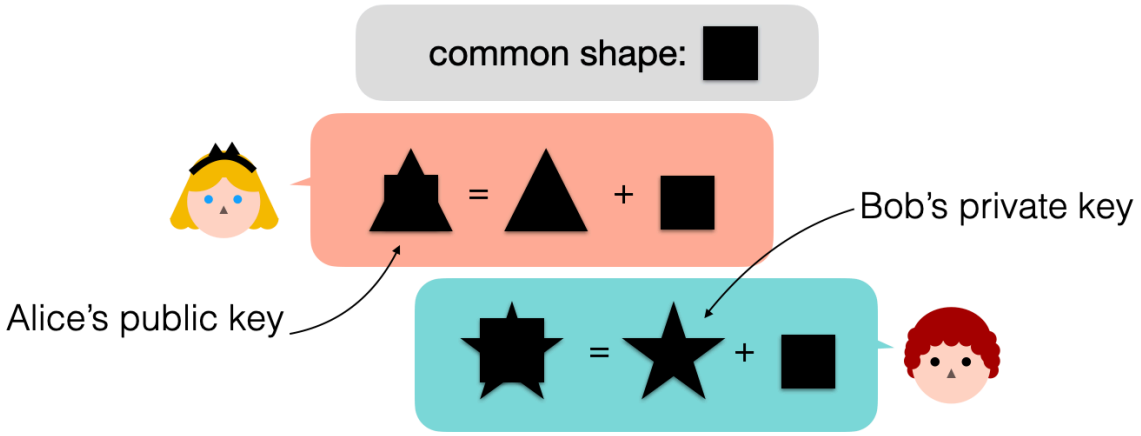


Figure 1.6 The second step of a Diffie-Hellman key exchange is to have both participants exchange their public keys. Participants derive their public keys by combining their private keys with the common shape.

We are now starting to see why this algorithm is called a public-key algorithm. It is because it requires a **key pair**: a private key and a public key.

The final step of the Diffie-Hellman key exchange algorithm is quite simple: Alice takes Bob's public key, and combines it with her private key. Bob does the same with Alice's public key, and combines it with his own private key. The result should now be the same on each side, in our example a shape combining a star, a square and a triangle (see figure 1.7).

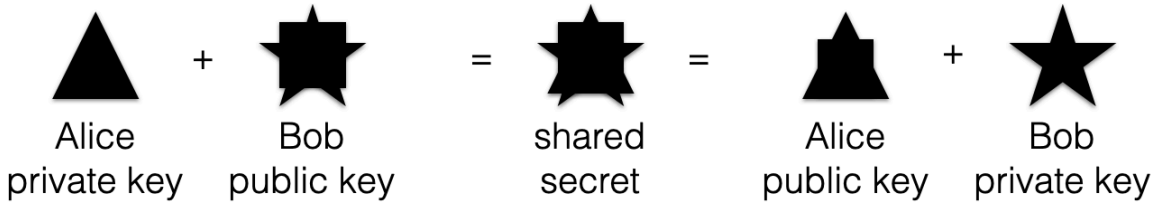


Figure 1.7 The final step of a Diffie-Hellman key exchange is to have both participants produce the same shared secret. To do this, Alice combines her private key with Bob's public key, and Bob combines his private key with Alice's public key. The shared secret cannot be obtained from solely observing the public keys.

It is now up to the participants of the protocol to make use of this **shared secret**. We will see several examples of this in this book, but the most obvious use case is to make use of it in an algorithm that requires a shared secret. For example, Alice and Bob could now use the shared secret as a key to encrypt further messages with a symmetric encryption primitive.

To recap:

1. Alice and Bob exchanged their public keys, which is masking their respective private keys.
2. With the other participant's public key, and their respective private key, they can compute a shared secret.

3. An adversary who observes the public keys being exchanged doesn't have enough information to compute the shared secret.

> **NOTE** In our example, the last point is easily bypassable. Indeed, without the knowledge of any private keys we can combine the public keys together to produce the shared secret. Fortunately, this is only a limitation of our analogy, but it works well enough for us to understand what a key exchange does.

In practice, a Diffie-Hellman key exchange is quite insecure. Can you take a few seconds to figure out why?

As Alice will accept any public key she receives as being Bob's public key, I could intercept the exchange and replace it with mine, which would allow me to impersonate Bob to Alice (and the same can be done to Bob). We say that a **man in the middle (MITM)** can successfully attack the protocol. How do we fix this? We will see in later chapters that we either need to augment this protocol with another cryptographic primitive, or we need to be aware in advance of what Bob's public key is.

But then, aren't we back to square one?

Previously Alice and Bob needed to know a shared secret, now Alice and Bob need to know their respective public keys. How do they get to know that? Is that a chicken-and-egg problem all over again? Well, kind of. As we will see, in practice public-key cryptography does not solve the problem of trust, but it simplifies its establishment (especially when the number of participants is large).

Let's stop here and move on to the next section, as you will learn more about key exchanges in chapter 5. We still have a few more asymmetric cryptographic primitives to uncover (see figure 1.8), to finish our tour of real-world cryptography.

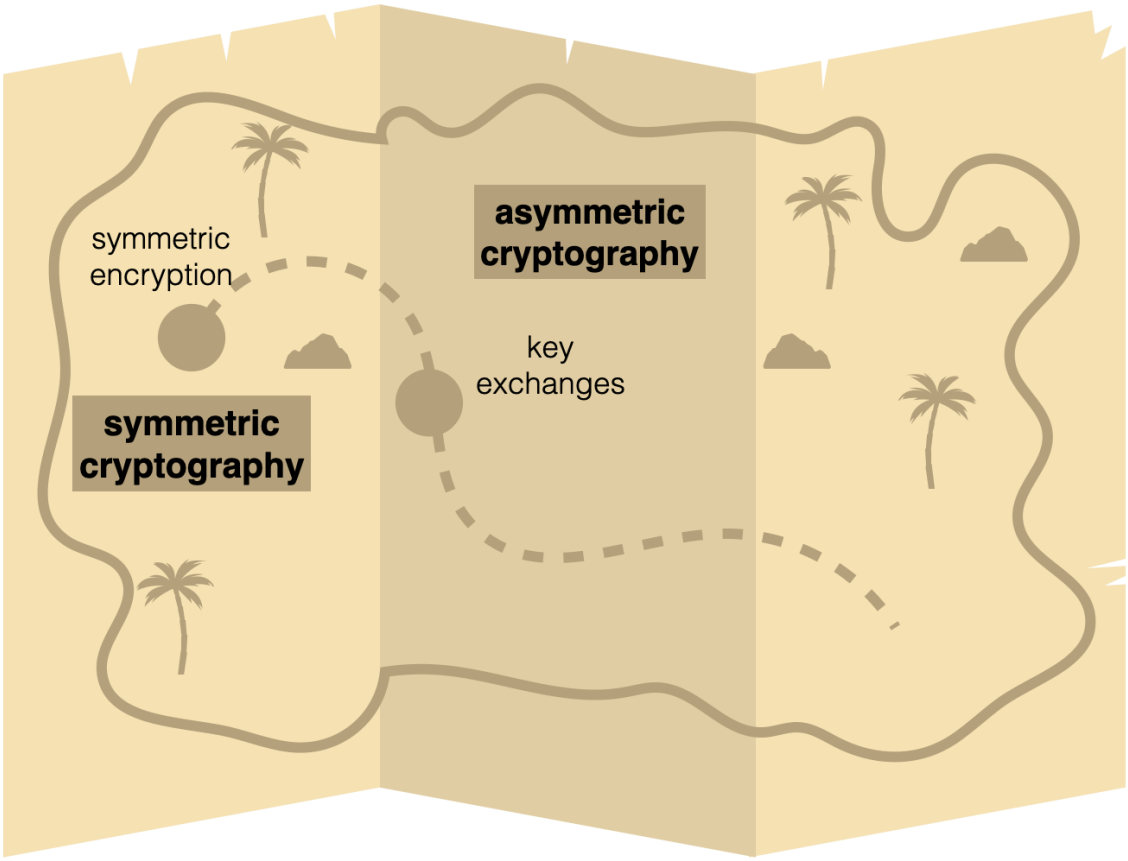


Figure 1.8 The cryptographic algorithms we have learned so far. Two large classes of cryptographic algorithms are symmetric cryptography (with symmetric encryption) and asymmetric cryptography (with key exchanges).

1.4.2 Asymmetric encryption, not like the symmetric one

The invention of the Diffie-Hellman key exchange algorithm was quickly followed by the invention of the **RSA** algorithm named after Ron Rivest, Adi Shamir, and Leonard Adleman. RSA contains two different primitives: a **public-key encryption algorithm** (or **asymmetric encryption**) and a **(digital) signature scheme**. Both primitives are part of the larger class of cryptographic algorithms called asymmetric cryptography. In this section we will explain what these primitives do, and how they can be useful.

The first one, **asymmetric encryption**, has a similar purpose to the symmetric encryption algorithm we've talked about previously: it allows one to encrypt messages in order to obtain confidentiality. Yet, unlike symmetric encryption, which had the two participants encrypt and decrypt messages with the same symmetric key, asymmetric encryption is quite different:

- it works with two different keys: a public key and a private key.
- it provides an asymmetric point of view: anyone can encrypt with the public key, but only the owner of the private key can decrypt messages.

Let's now use a simple analogy to explain how one can use asymmetric encryption. We start with our friend Alice again, who holds a private key (and its associated public key). Let's picture her public key as an open chest that she releases to the public for anyone to use (see figure 1.9).

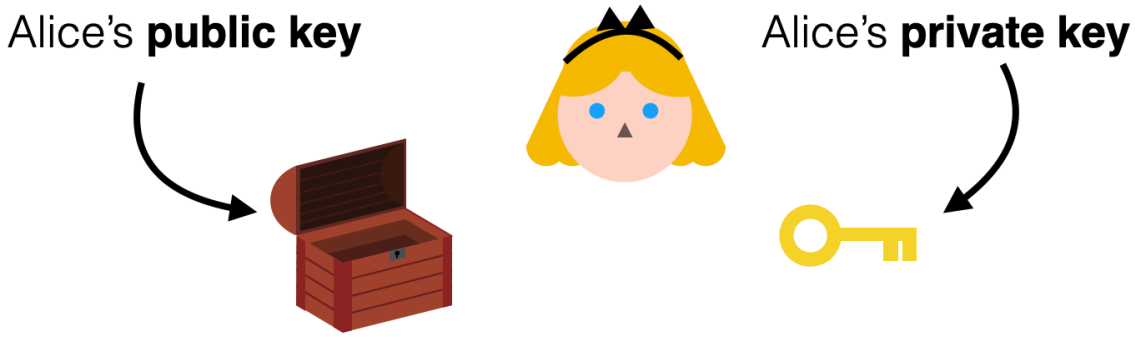


Figure 1.9 To use asymmetric encryption, Alice needs to first publish her public key (represented as an open box here). Now, anyone can use the public key to encrypt messages to her. And she should be able to decrypt them using the associated private key.

Now, you and I and everyone who wants can encrypt a message to her using her public key. In our analogy, imagine that you would insert your message into the open chest and then close it. Once the chest is closed, nobody but Alice should be able to open it. The box effectively protects the secrecy of the message from observers on the way.

The closed box (or encrypted content) can then be sent to Alice, and she can use her private key (only known to her, remember) to decrypt it (see figure 1.10).

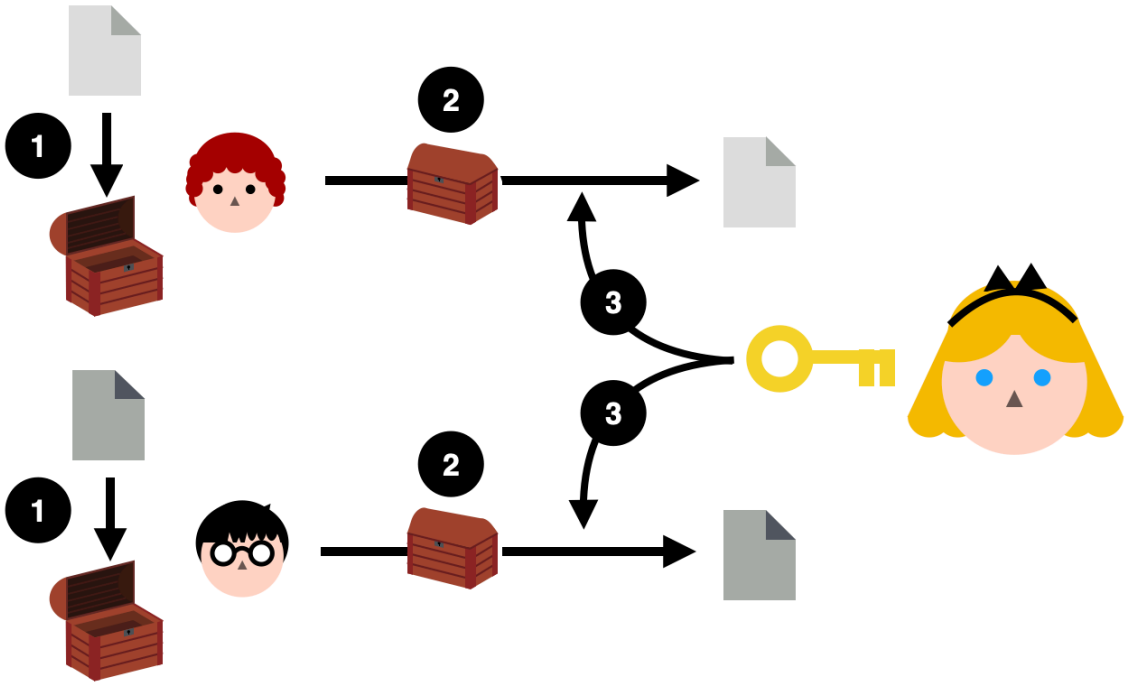


Figure 1.10 Asymmetric Encryption: (1) anyone can use Alice's public key to encrypt messages to her. (2) After receiving them, (3) she can decrypt the content using her associated private key. Nobody is able to observe the messages directed to Alice while they are being sent to her.

Let's summarize in figure 1.11 the cryptographic primitives we have learned so far. We are only missing one more to finish our tour of real-world cryptography!

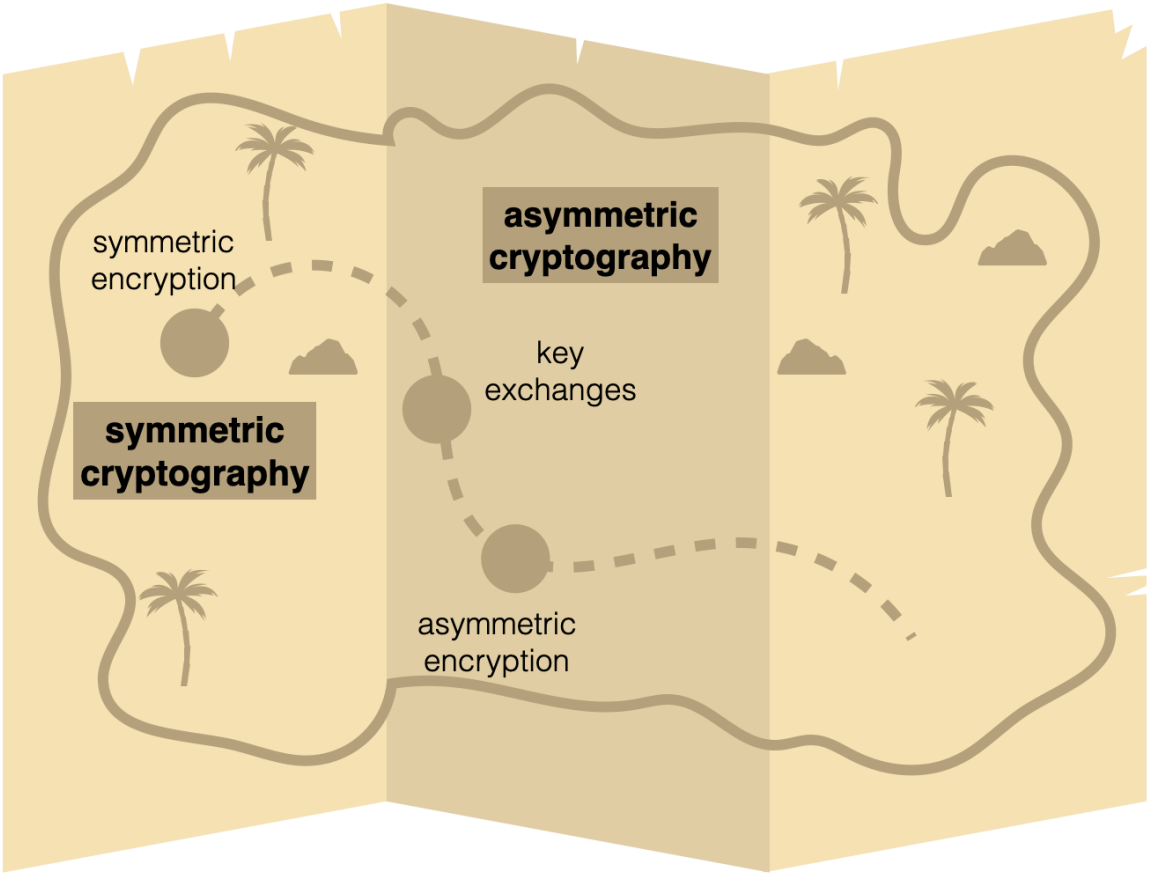


Figure 1.11 The cryptographic algorithms we have learned so far. Two large class of cryptographic algorithms are symmetric cryptography (with symmetric encryption) and asymmetric cryptography (with key exchanges and asymmetric encryption).

1.4.3 Digital signatures, just like your pen-and-paper signatures

We've seen that RSA provides an **asymmetric encryption** algorithm, but as we've mentioned earlier, it also provides a **digital signature** algorithm.

> **NOTE** **The word "RSA" is often used to mean either RSA encryption or RSA signatures depending on context, which is always a great source of confusion. You've been warned.**

The invention of this digital signature cryptographic primitive has been of immense help to set up trust between the Alices and Bobs of our world. It is similar to real signatures, you know, the one you are required to sign on a contract when you're trying to rent an apartment, for example. "What if they forge my signature?" you may ask, and indeed real signatures don't provide much security in the real-world. On the other hand, cryptographic signatures can be used in the same kind of way, but provide a cryptographic certificate with your name on it. Your cryptographic signature is **unforgeable**, and can easily be verified by others. Pretty useful compared to the archaic signatures you used to write on checks!

In figure 1.12, we can imagine a protocol where Alice wants to show David that she trusts Bob. This is a typical example of how to establish trust in a multi-participant setting, and how asymmetric cryptography can help. By signing a piece of paper containing "*I, Alice, trust Bob*," Alice can take a stance and notify David that Bob is to be trusted. If David already trusts Alice and her signature algorithm, then he can choose to trust Bob in return.

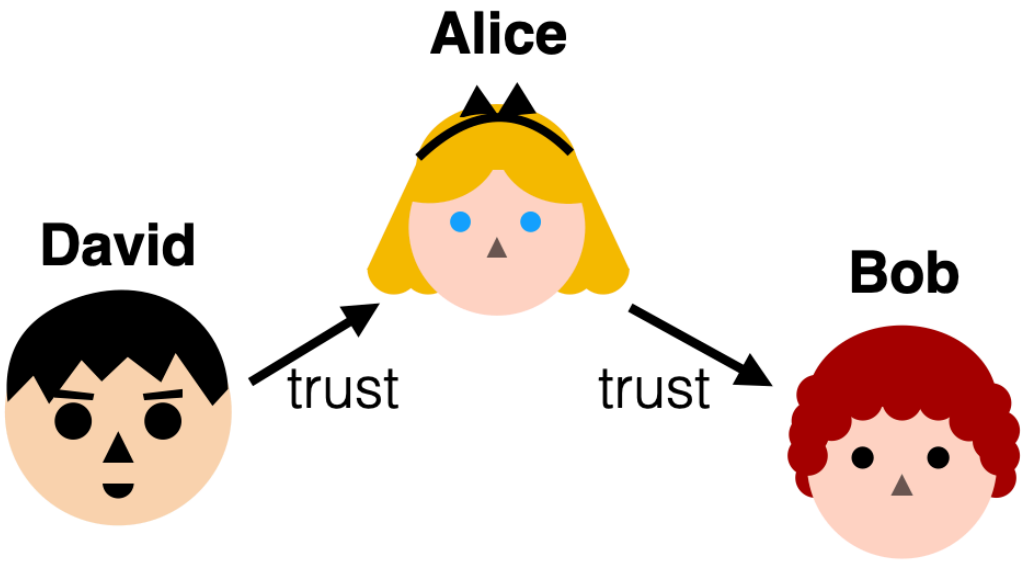


Figure 1.12 David already trusts Alice. Since Alice trusts Bob, can David safely trust Bob as well?

In more detail, Alice can use the **RSA signature scheme** and her **private key** to **sign** the message "*I, Alice, trust Bob*." This generates a signature that should look like random noise (see figure 1.13).

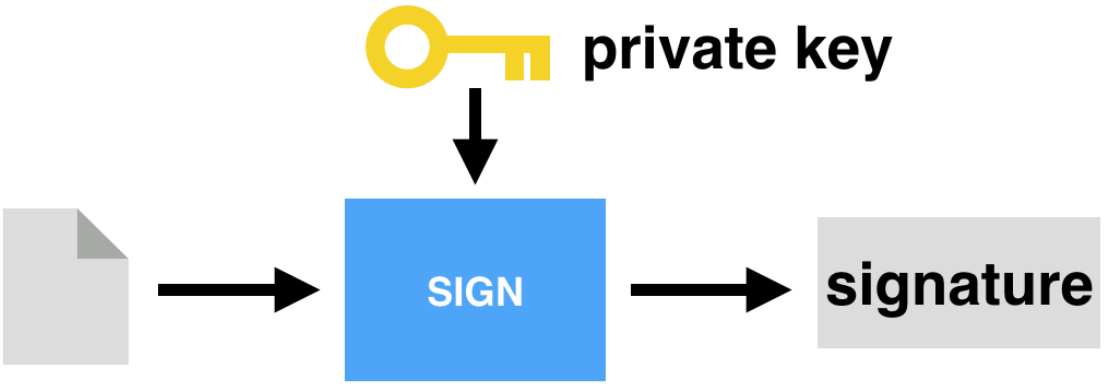


Figure 1.13 To sign a message, Alice uses her private key and generates a signature.

Anyone can then **verify the signature**, by combining:

- Alice's **public key**.

- The **message** that was signed.
- The **signature**.

The result is either `true` (the signature is valid) or `false` (the signature is invalid) as can be seen in figure 1.14.

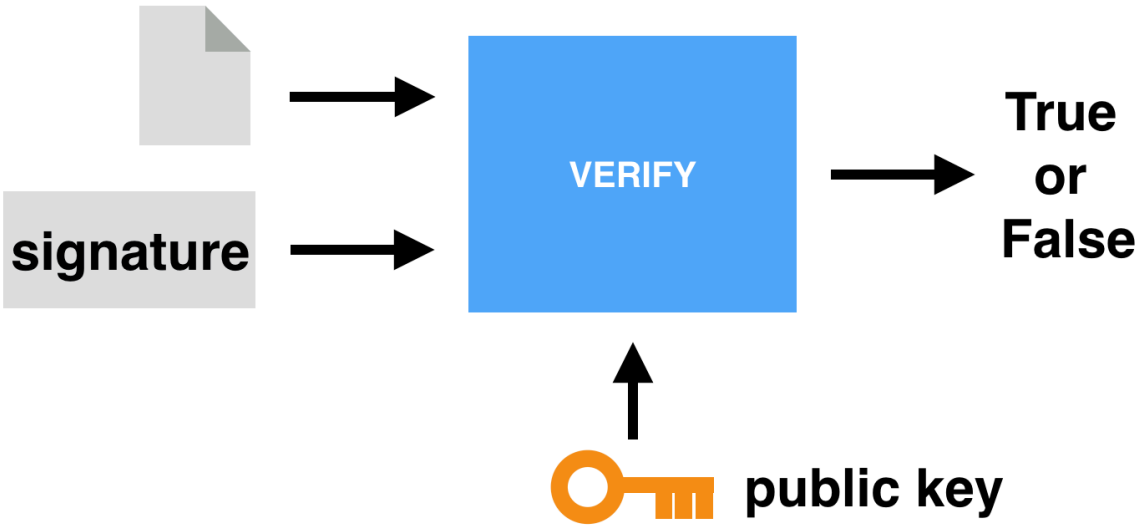


Figure 1.14 To verify a signature from Alice, one also needs the message signed and Alice's public key. The result is either validating the signature, or invalidating it.

We have now learned about three different asymmetric primitives:

- Key exchange with Diffie-Hellman.
- Asymmetric encryption.
- Digital signatures with RSA.

These three cryptographic algorithms are the most known and commonly used primitives in asymmetric cryptography. It might not be totally obvious how these can help to solve real-world problems, but rest assured, they are used every day by many applications to secure things around them.

It is time to complete our picture with all the cryptographic algorithms we've learned about so far. See figure 1.15.

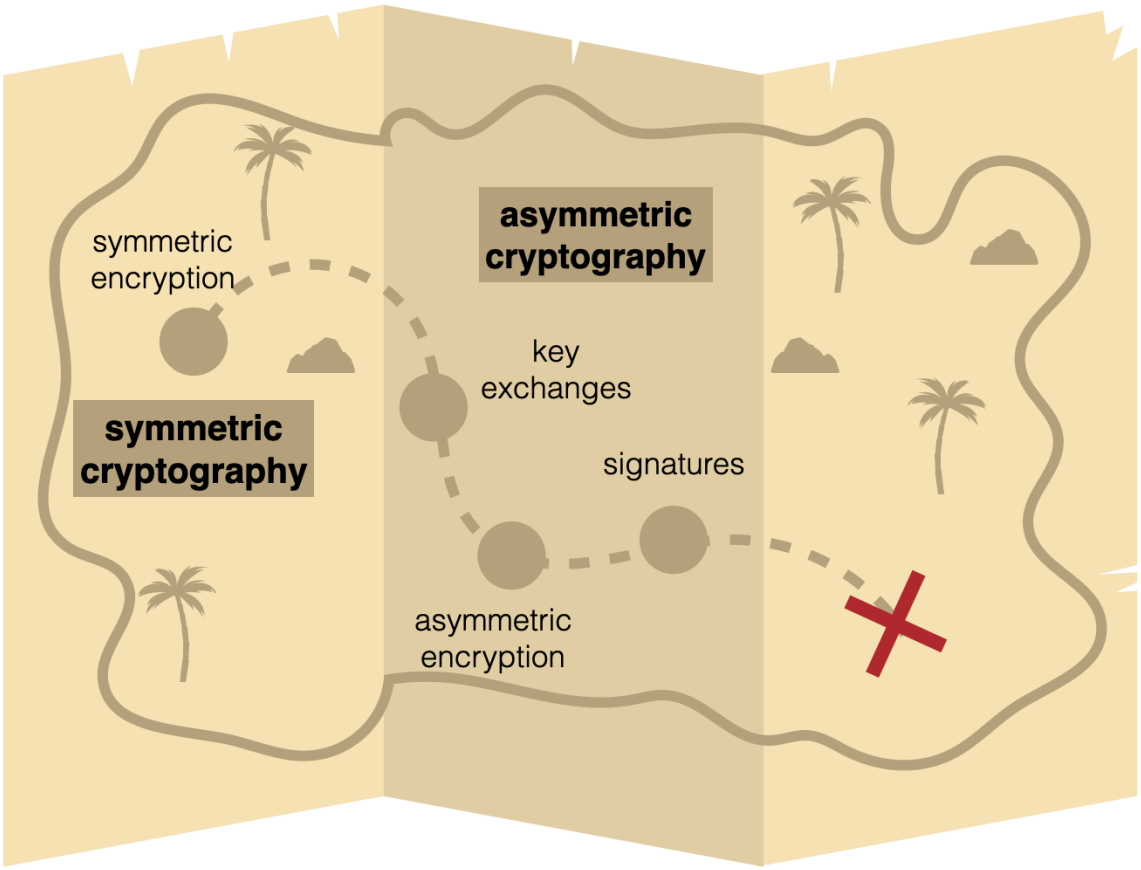


Figure 1.15 The symmetric and asymmetric algorithms we have learned so far.

1.5 Classifying and abstracting cryptography

In the previous section, we have surveyed two large classes of algorithms:

- **Symmetric cryptography** (or secret-key cryptography). A single secret is used. If several participants are aware of the secret, it is called a shared secret.
- **Asymmetric cryptography** (or public-key cryptography). Participants have an asymmetrical view of the secrets. For example, some will have knowledge of a public key, while some will have knowledge of both a public and private key.

Symmetric and asymmetric cryptography are not the only two categories of primitives in cryptography, and it's quite hard to classify the different subfields. But yet, as you will realize, a large part of our book is about (and makes use of) symmetric and asymmetric primitives. This is because a large part of what is useful in cryptography, nowadays, is contained in these subfields.

Another way of dividing cryptography can be:

- **Math-based constructions**. These are constructions that rely on mathematical problems like factoring numbers. (The RSA algorithm for digital signatures and asymmetric encryption is an example of such a construction.)

- **Heuristic-based constructions**. These are the constructions that rely on observations and statistical analysis by cryptanalysts. (AES for symmetric encryption is an example of such a construction.)

There is also a speed component to this categorization, as mathematic-based constructions are often much slower than heuristic-based constructions. To give you an idea, symmetric constructions are most often based on heuristics (what seems to be working), while most asymmetric constructions are based on mathematical problems (what is thought to be hard).

It is hard for us to rigorously categorize all of what cryptography has to offer. Indeed, every book or course on the subject gives different definitions and classifications. In the end, these distinctions are not too useful for us, as we will see most of the cryptographic primitives as unique tools that make unique **security claims**. Many of these tools can in turn be used as building blocks to create protocols. It is thus essential to understand how each of these tools work, and what kind of security claims they provide, in order to understand how they provide security in the protocols around us. For this reason, the first part of this book will go through the most useful cryptographic primitives and their security properties.

A lot of the concepts in the book can be quite complicated the first time around. But like everything, the more we read about them, and the more we see them in context, the more natural they become. The more we abstract them. The role of this book is to help you to create these abstractions, to allow you to create a mental model of what these constructions do, and how they can be combined together to produce secure protocols. I will often talk about the interface of constructions, and give real-world examples of usage and composition.

The definition of cryptography used to be simple: Alice and Bob want to exchange secret messages. It isn't anymore. What cryptography is nowadays, is quite complex to describe, and has grown organically around discoveries, breakthroughs and practical needs. At the end of the day, **cryptography is what helps to augment a protocol, in order to make it work in adversarial settings**.

To understand exactly how cryptography can help, the set of goals that these protocols want to achieve is what matters to us. That's the useful part. Most of the cryptographic primitives and protocols we'll learn about in this book provide one or two of the following properties:

- **Confidentiality**. It's about masking and protecting some information from the wrong eyes. For example, encryption masks the messages in transit.
- **Authentication**. It's about identifying who we are talking to. For example, this can be helpful in making sure that messages we receive indeed come from Alice.

Of course, it's still a heavy simplification of what cryptography can provide. In most cases, the details are in the security claims of the primitives. Depending on how a cryptographic primitive is used in a protocol, it will achieve different security properties.

Throughout this book, we will learn new cryptographic primitives and how they can be combined to expose security properties like confidentiality and authentication. For now, appreciate the fact that cryptography is about providing insurances to a protocol in adversarial settings. While the "adversaries" are not clearly defined, we can imagine that they are the ones who attempt to break your protocol: a participant, an observer, a man in the middle. They reflect what a real life adversary could be. Because eventually, cryptography is a practical field made to defend against bad actors in flesh and bones and bits.

1.6 Theoretical cryptography versus real-world cryptography

In 1993, Schneier releases Applied Cryptography, a book targeting developers and engineers who want to build applications that involve cryptography. Circa 2012, Kenny Paterson and Nigel Smart start an annual conference called Real World Crypto, targeting the same crowd and taking place every year. But what do applied cryptography and real-world cryptography refer to? Is there more than one type of cryptography?

To answer the question, we have to start by defining **theoretical cryptography**, the cryptography that cryptographers and cryptanalysts work on. These crypto people are mostly from the academia, working in universities, but sometimes from the industry or in specific departments of the government. They research everything and anything in cryptography. Results are shared internationally through publications and presentations in journals and conferences. Yet, not everything they do is obviously useful or practical. Often, no "proof of concept" or code is released, and it wouldn’t make sense anyway as no computer is powerful enough to run their research. Having said that, theoretical cryptography does sometimes become so useful and practical that it makes its way to the other side.

The other side is the the world of applied cryptography, or **real-world cryptography**. It is the foundation of the security you find in all applications around you. Although it often seems like it’s not there, almost transparent, it is here when you log into your bank account on the internet, it is with you when you message your friends, it helps protect you when you lose your phone. It is ubiquitous, because unfortunately attackers are everywhere and actively do try to observe and harm our systems. Practitioners are usually from the industry, but will sometimes vet algorithms and design protocols with the help of the academia community. Results are often shared through conferences, blog posts, and open-source software.

Real-world cryptography usually cares deeply about real-world considerations: what is the exact level of security provided by an algorithm? How long does it take to run the algorithm? What is the size of the inputs and outputs required by the primitive? Real-world cryptography is, as you might have guessed, the subject of this book. While theoretical cryptography is the subject of other books, we will still take a peek at what is brewing there in the last chapters of this book. Be prepared to be amazed, as you might catch a glance of the real-world cryptography of tomorrow.

Now you might wonder: how do developers and engineers choose what cryptography to use for their real-world applications?

1.7 From theoretical to practical: choose your own adventure

> *Sitting on top are cryptanalysts who propose and solve hard mathematical problems […] and at the bottom are software engineers who want to encrypt some data.*
>
> *– Thai Duong So you want to roll your own crypto? (2020)*

In all those years I've spent studying and working with cryptography, I've never noticed a single pattern in which a cryptographic primitive ends up being chosen to be used in real-world applications. Things are pretty chaotic, before a theoretical primitive gets to be adopted there's a long list of people who get to handle the primitive and shape it into something consumable and sometimes safer for the public at large. So how can I even explain that to you… Have you heard of Choose Your Own Adventure? It's an old book series that got you to pick how you wanted to step through the story. The principle was simple: you read the first section of the book; at the end of the section, the book lets you decide on the path forward by giving you different options; each option was associated to a different section number that you could skip directly to if you so chose. So I've done the same! Start by reading the next paragraph, and follow the direction it gives you.

Where it all begins. Who are you? Are you Alice, a cryptographer? Are you David, working in the private industry and in need of a solution to your problems? Or are you Eve, working in a government branch and preoccupied by cryptography?

- You're Alice, go to step 1.
- You're David, go to step 2.
- You're Eve, go to step 3.

Step 1 - Researchers gotta research. You're a researcher working in a university, or in the research team of a private company or a non-profit, or in a government research organization like the NIST or the NSA. As such, your funding can come from different places and might incentivize you to research different things.

- You invent a new primitive, go to step 4.
- You invent a new construction, go to step 5.
- You start an open competition, go to step 6.

Step 2 - The industry has a need. As part of your job, something comes up, and you are in need of a new standard. For example, the Wi-Fi Alliance is a non-profit funded by interested companies to produce the set of standards around the Wi-Fi protocol. Another example are banks, that got together to produce the Payment Card Industry Data Security Standard (PCI-DSS) which enforces algorithms and protocols to use if you deal with credit card numbers.

- You decide to fund some much-needed research, go to step 1.
- You decide to standardize a new primitive or protocol, go to step 5.
- You start an open competition, go to step 6.

Step 3 - A government has a need. You're working for your country's government, and you need to push out some new crypto. For example, the NIST is tasked with publishing the Federal Information Processing Standards (FIPS) which mandates what cryptographic algorithms can be used by companies that deal with the US government. While many of these standards were success stories, and people tend to have a lot of trust in standards being pushed by government agencies, there is unfortunately a lot to say about failures. In 2013, it was discovered following revelations from Edward Snowden, that the NSA had purposefuly and successfully pushed for the inclusion of backdoors algorithms in standards (see "Dual EC: A Standardized Back Door" by Bernstein et al.). which included a secret switch that allowed the NSA and only the NSA to predict your secrets). These backdoors can be thought as magic passwords that allow the government (and only them, suposedly) to subvert your encryption. Following this, the cryptographic community has lost a lot of confidence in standards and suggestions coming from government bodies. Recently, in 2019, it was found that the russian standard GOST had been victim of the same treatment.

> *Cryptographers have long suspected that the agency planted vulnerabilities in a standard adopted in 2006 by the National Institute of Standards and Technology and later by the International Organization for Standardization, which has 163 countries as members. Classified N.S.A. memos appear to confirm that the fatal weakness, discovered by two Microsoft cryptographers in 2007, was engineered by the agency. The N.S.A. wrote the standard and aggressively pushed it on the international group, privately calling the effort "a challenge in finesse.*
>
> *– New York Times N.S.A. Able to Foil Basic Safeguards of Privacy on Web (2013)*

- You fund some research, go to step 1.
- You organize an open competition, go to step 6.
- You push for the standardization of a primitive or protocol that you're using, go to step 7.

Step 4 - A new concept is proposed. As a researcher, you manage to do the impossible, you invent a new concept. Sure someone already thought about encryption, but there are still new primitives being proposed every year in cryptography. Some of them will prove to be impossible to realize, and some of them end up being solvable. Maybe you have an actual construction as part of your proposal, or maybe you'll have to wait to see if someone can come up with something that works.

- Your primitive gets implemented, go to step 5.
- Your primitive ends up being impossible to implement, go back to the beginning.

Step 5 - A new construction or protocol is proposed. A cryptographer, or team of

cryptographers, proposes a new algorithm that instantiates a concept. For example, AES is an instantiation of an encryption scheme. AES was initially proposed by Vincent Rijmen and Joan Daemen (who named their construction as a constraction of their names, Rijndael). What's next?

- Someone builds on your construction, go to step 5.
- Your partake in an open competition and win! Go to step 6.
- There's a lot of hype for your work, you're getting a standard! Go to step 7.
- You decide to patent your construction, go to step 8.
- You, or someone else, decides that it'll be fun to implement your construction. Go to step 9.

Step 6 - An algorithm wins a competition. The process cryptographers love the most is an open competition! For example, the Advanced Encryption Standard (AES) was a competition that invited researchers from all over the world to compete. After dozens of submissions and rounds of analysis and help from cryptanalysts (which can take years), the list was reduced to a few candidates (in the case of AES, a single one), which then moved to become standardized.

- You got lucky, after many years of competition your construction won! Go to step 7
- Unfortunately, you lost. Go back to the start.

Step 7 - An algorithm or protocol is standardized. A standard is usually published by a government or by a standardization body. The aim is to make sure that everyone is on the same page, so as to maximalize interoperability. For example, the NIST (a US government organization) regularly publishes cryptographic standards. A well-known standardization body in cryptography is the Internet Engineering Task Force (IETF), which is behind many standards on the internet (like TCP, UDP, TLS, and so on), and that you will hear about a lot in this book. Standards in the IETF are called **Request For Comment (RFC)**, and can be written by pretty much anyone who wants to write a standard.

> *To reinforce that we do not vote, we have also adopted the tradition of "humming": When, for example, we have face-to-face meetings and the chair of the working group wants to get a "sense of the room", instead of a show of hands, sometimes the chair will ask for each side to hum on a particular question, either "for" or "against".*
>
> *– RFC 7282 On Consensus and Humming in the IETF (2014)*

Sometimes, a company publishes a standard directly. For example, RSA Security LLC (funded by the creators of the RSA algorithm), released a series of 15 documents called the Public Key Cryptography Standards (PKCS), in order to legitimize algorithms and techniques the company was using at the time. Nowadays, this is pretty rare and a lot of companies will go through the IETF to standardize their protocols or algorithms as an RFC, instead of a custom document.

- Your algorithm or protocol gets implemented, go to step 9.
- Nobody cares about your standard, go back to the start.

Step 8 - A patent expires. A patent in cryptography usually means that nobody will use the algorithm. Once the patent expired, it is not uncommon to see a renewed interest in the primitive. The most popular example is probably Schnorr signatures, which were the first contender to become the most popular signature scheme until Schnorr himself patented the algorithm in 1989. This led to the NIST standardizing a poorer algorithm (DSA), which became (at the time) the go-to signature scheme. The patent over Schnorr signatures has expired in 2008, and the algorithm has since started regaining popularity.

- It's been too long, your algorithm will be forever forgotten. Go back to the beginning.
- Your constructions inspires many more constructions to get invented on top of it, go to step 5.
- Now people want to use your construction, but not before it's standardized for real. Go to step 7.
- Some developers are implementing your algorithm! Go to step 9.

Step 9 - A construction or protocol gets implemented. Implementers have the hard task to not only decipher a paper or a standard (although standards are supposed to target implementers, "supposed"), but they also must make their implementations easy and safe to use. This is not always an easy task, as many devastating bugs can arise in the way cryptography is used.

- Someone decides it is time for these implementations to be backed by a standard. It's embarassing without one. Go to step 7.
- Hype is raining on your cryptographic library! Go to step 10.

Step 10 - A developer uses a protocol or primitive in an application. A developer has a need, and your cryptographic library seems to solve it. Easy as!

- The primitive solves the need, but they don't have a standard… Not great. Go to step 7.
- I wish this was written in my programming language. Go to step 9.
- I misused the library, or the construction is broken. Game over.

You got it: there's many means for a primitive to go real-world. The best way involves many years of analysis, a good standard, and good libraries; A bad way involves a bad algorithm with a bad implementation. In figure 1.16 I illustrate the preferred path.

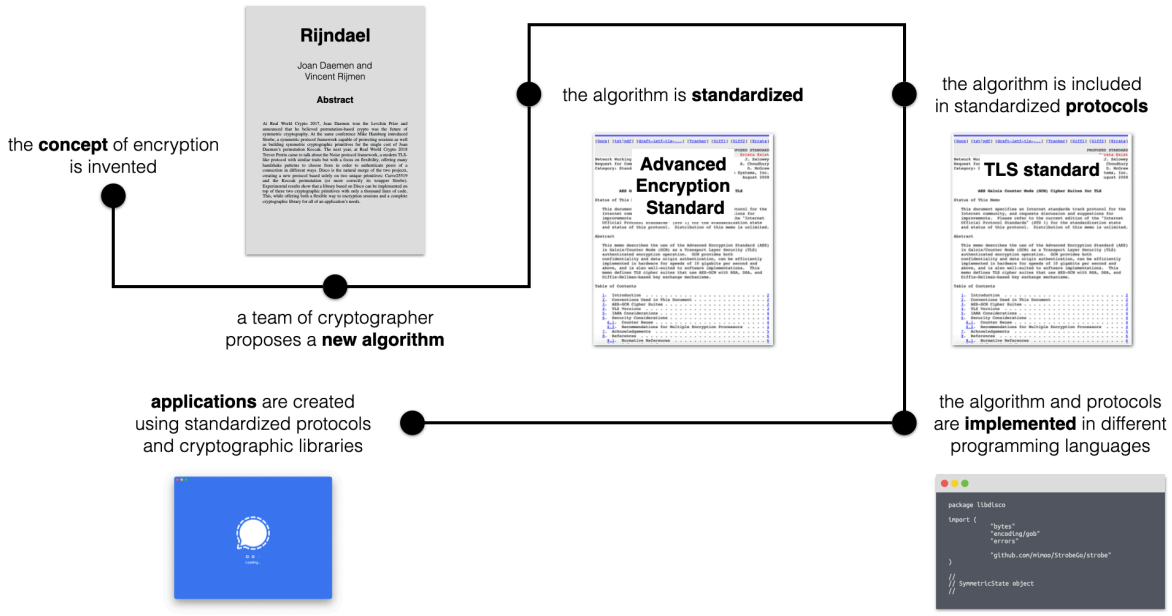


Figure 1.16 The ideal lifecycle for a cryptographic algorithm: it starts when cryptographers instantiate a concept in a whitepaper. For example, AES is an instantiation of the concept of symmetric encryption (there are many more symmetric encryption algorithms out there). A construction can then be standardized: everybody agree to implement it a certain way to maximize interoperability. Then support is created by implementing the standard in different languages.

1.8 A word of warning

Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can't break.

– Bruce Schneier Memo to the Amateur Cipher Designer (1998)

I must warn you, the art of cryptography is difficult to master and it would be unwise to build complex cryptographic protocols once you're done with this book.

This journey should enlighten you, show you what is possible, and show you how things work, but it will not make you a master of cryptography. This book is not the holy grail. Indeed, the last pages of this book will have you go through the most important lesson: do not go alone on a real adventure. Dragons can kill, and you need some support to accompany you in order to defeat them. In other words, cryptography is complicated, and this book alone does not permit you to abuse what you learn. To build complex systems, experts who have studied their trade for years are required. Instead, what you will learn is to recognize when cryptography should be used or if something seems fishy, what cryptographic primitives and protocols are available to solve the issues you're facing, and how all these cryptographic algorithms work under the surface.

Now that you've been warned, go to the next chapter.

1.9 Summary

- A protocol is a step by step recipe where multiple participants attempt to achieve something, like exchanging confidential messages.
- Cryptography is about augmenting protocols to secure them in adversarial settings. It often requires secrets.
- A cryptographic primitive is a type of cryptographic algorithm. For example, symmetric encryption is a cryptographic primitive, while AES is a specific symmetric encryption algorithm.
- One way to classify the different cryptographic primitives is to split them into two types: symmetric and asymmetric cryptography: symmetric cryptography uses a single key (as you've seen with symmetric encryption), while asymmetric cryptography makes use of different keys (as you've seen with key exchanges, asymmetric encryption, and digital signatures).
- Cryptographic properties are hard to classify, but they often aim to provide one of these two properties: authentication or confidentiality. Authentication is about verifying the authenticity of something or someone, while confidentiality is about the privacy of data or identities.
- real-world cryptography matters because it is ubiquitous in technological applications, while theoretical cryptography is often less useful in practice.
- Most of the cryptographic primitives contained in this book were agreed on after long standardization processes.
- Cryptography is complicated and there are many dangers in implementing or using cryptographic primitives.

2 Hash functions

This chapter covers

- Hash functions and their security properties.
- The wildly adopted hash functions in use today.
- What other types of hashing there exist.

I have talked about cryptographic primitives in the previous chapter, they are constructions that achieve specific security properties and form the building blocks of cryptography. In the first part of this book you will learn about several important ones. The first one I'll talk about is the **hash function**. Informally, it is a tool that when given an input will produce a unique identifier tied to that input. A hash function is rarely used on its own but can be found everywhere in cryptography. For example, we will rarely see digital signatures not making use of hash functions (and some signature schemes even go as far as being built solely using hash functions).

2.1 What is a hash function?

In front of you a download button is taking a good chunk of the page. You can read the letters *DOWNLOAD*, and clicking on them seems to redirect you to a different website containing the file. Below it, lies a long string of unintelligible letters:

```
f63e68ac0bf052ae923c03f5b12aedc6cca49874c1c9b0ccf3f39b662d1f487b
```

It is followed by what looks like an acronym of some sort: `sha256sum`. Sounds familiar? You've probably downloaded something in your past life that was also accompanied with such an odd string.

Figure 2.1 A webpage linking to an external website containing a file. The external website cannot modify the content of the file, because the first page provides a hash or digest of the file, which ensures the integrity over the downloaded file.

If you've ever wondered what was to be done with it:

1. Click on the button to download the file.
2. Use the SHA-256 algorithm to *hash* the downloaded file.
3. Compare the output (the digest) with the long string displayed on the webpage.

This allows you to verify that you've downloaded the right file.

> **NOTE** **The output of a hash function is often named a digest or a hash. I will use the two words interchangeably throughout this book. Note that others might call it a checksum or a sum, which I will avoid as it is primarily used by non-cryptographic hash functions and could lead to more confusion. Just keep that in mind when different codebases or documents use different terms.**

To try hashing something, you can use the popular **OpenSSL** library. It offers a multi-purpose Command Line Interface that comes by default in a number of systems, including macOS. For example, this can be done by opening the terminal and writing the following line:

```
$ openssl dgst -sha256 downloaded_file
f63e68ac0bf052ae923c03f5b12aedc6cca49874c1c9b0ccf3f39b662d1f487b
```

We used SHA-256 (a hash function) to transform the input (the file) into a unique identifier (the digest). What do these extra steps provide? **Integrity** and **Authenticity**. It tells you that what you downloaded is indeed the file you were meant to download.

All of this works thanks to a security property of the hash function called **second pre-image resistance**. This math-inspired term means that from the long output of the hash function `f63e`…, you cannot find another file that will "hash" to the same output `f63e`…. In practice it means that this digest is closely tied to the file you're downloading, and that no attacker should be able to fool you by giving you a different file.

> **NOTE** **By the way, the long output string `f63e`… represents binary data displayed in hexadecimal (a base 16 encoding using numbers from `0` to `9` and letters from `a` to `f` to represent several bits of data). We could have displayed the binary data with 0s and 1s (base 2) but it would have taken more space. Instead, the hexadecimal encoding allows us to write 2 alphanumeric characters for every 8 bits (1 byte) encountered. It is somewhat readable by humans and takes less space. There are other ways to encode binary data for human consumption, but the two most widely used encodings are hexadecimal and base64. The larger the base, the less space it takes to display a binary string, but at some point we run out of human-readable characters :)**

Note that this long digest is controlled by the owner(s) of the webpage, and it could easily be replaced by anyone who can modify the webpage. (If you are not convinced about this take a moment to think about it.) This means that we need to trust the page that gave us the digest, its owners and the mechanism used to retrieve the page (while we don't need to trust the page that gave us the file we downloaded). In this sense, **the hash function alone does not provide integrity**. The integrity and authenticity of the downloaded file comes from the digest combined with the trusted mechanism which gave us the digest (**HTTPS** in this case, we will talk about HTTPS in chapter 9 but for now imagine that it magically allows you to communicate securely with a website).

Back to our hash function, which can be visualized as a black box in figure 2.2 which takes a single input, and gives out a single output.

Figure 2.2 A hash function takes an arbitrary-length input (a file, a message, a video, and so on) and produces a fixed-length output (for example, 256 bits for SHA-256). Hashing the same input will produce the same digest or hash.

The **input** of this function can be of any size. It can even be empty.

The **output** is always of the same length and **deterministic**: it always produces the same result if given the same input. In our example, SHA-256 always provides an output of 256 bits (32 bytes) which is always encoded as 64 alphanumeric characters in hexadecimal. One major property of a hash function is that one cannot revert the algorithm, meaning that one shouldn't be able to find the input from just the output. We say that hash functions are **one-way**.

To illustrate how a hash function works in practice, we hash different inputs with the SHA-256 hash function using the same OpenSSL Command Line Interface:

```
$ echo -n "hello" | openssl dgst -sha256   ❶
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824   ❶
$ echo -n "hello" | openssl dgst -sha256   ❶
2cf24dba5fb0a30e26e83b2ac5b9e29e1b161e5c1fa7425e73043362938b9824   ❶
$ echo -n "hella" | openssl dgst -sha256   ❷
70de66401b1399d79b843521ee726dcec1e9a8cb5708ec1520f1f3bb4b1dd984   ❷
$ echo -n "this is a very very very very very very very very very very
very long sentence" | openssl dgst -sha256   ❸
009e286e0261a8d0eca95649cf795db3572c515fe9dc7e319ece5af8f133637a   ❸
```

❶ Hashing the same input produces the same result.

❷ A tiny change in the input completely changes the output.

❸ The output is always of the same size, no matter the input size.

In the next section, we will see what are the exact security properties of hash function.

2.2 Security properties of a hash function

Hash functions in applied cryptography are constructions that were commonly defined to provide three specific security properties. This definition has changed over time, as we will see in the next sections. But for now, let's define these three strong foundations that makes a hash function. This is important as you need to understand where hash functions can be useful, and where they will not work.

The first one is **pre-image resistance**. This property ensures that no one should be able to reverse the hash function in order to, given an output, recover the input. In figure 2.3 we

illustrate this one-wayness by imagining that our hash function is like a blender making it impossible to recover the ingredients from the produced smoothie.

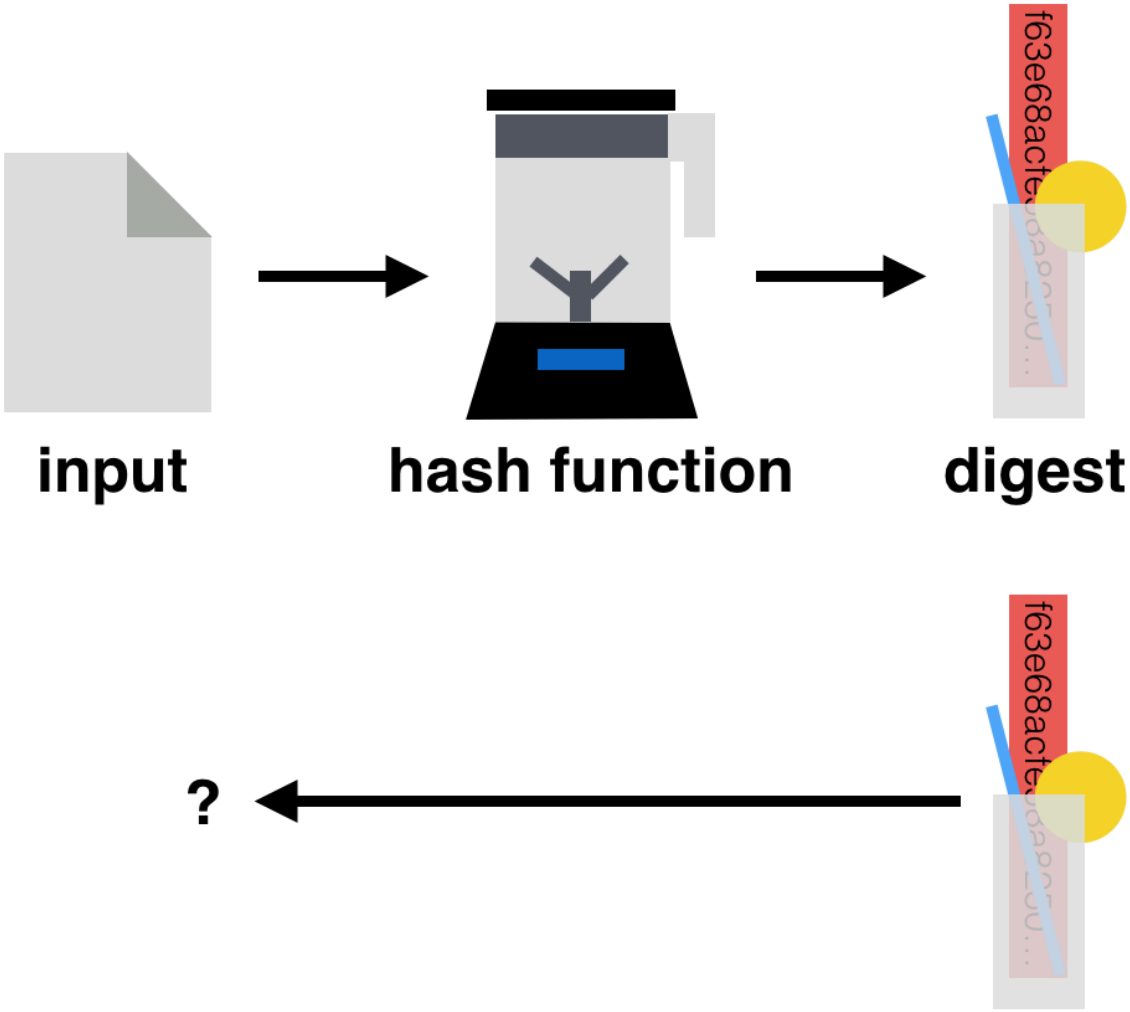


Figure 2.3 Given the digest produced by a hash function (represented as a blender here), it is impossible (or technically so hard we assume it should never happen) to reverse it and find the original input used. This security property is called pre-image resistance.

> **WARNING** Is this true if your input is small? Let's say that it's either "oui" or "non," then it is easy for someone to hash all the possible 3-letter words and find out what the input was. What if your input space is small? Meaning that you always hash variants of the sentence "I will be home on Monday at 3am." Here one who can predict this, but does not know exactly the day of the week or the hour, can still hash all possible sentences until it produces the correct output. As such, this first pre-image security property has an obvious caveat: you can't hide something that is too small or is predictable.

The second one is **second pre-image resistance**. We've already seen this security property when

we wanted to protect the integrity of a file. The property says the following: if I give you an input and the digest it hashes to, you should not be able to find a **different input** that hashes to the same digest. This is illustrated in figure 2.4.

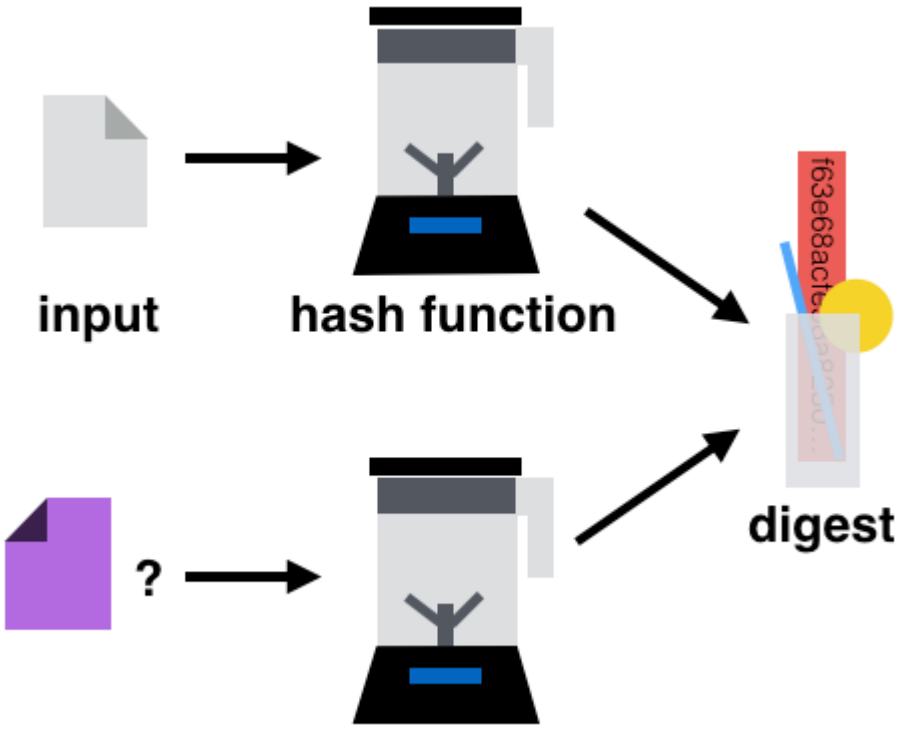


Figure 2.4 Considering an input and its associated digest, one should never be able to find a different input that hashes to the same output. This security property is called second pre-image resistance.

Note that **we do not control the first input**, this emphasis is important to understand the next security property.

Finally, the third property is **collision-resistance**. It guarantees that no one should be able to produce two different inputs that hash to the same output (as seen in figure 2.5). Here an attacker can choose the two inputs, unlike the previous property that fixes one of the inputs.

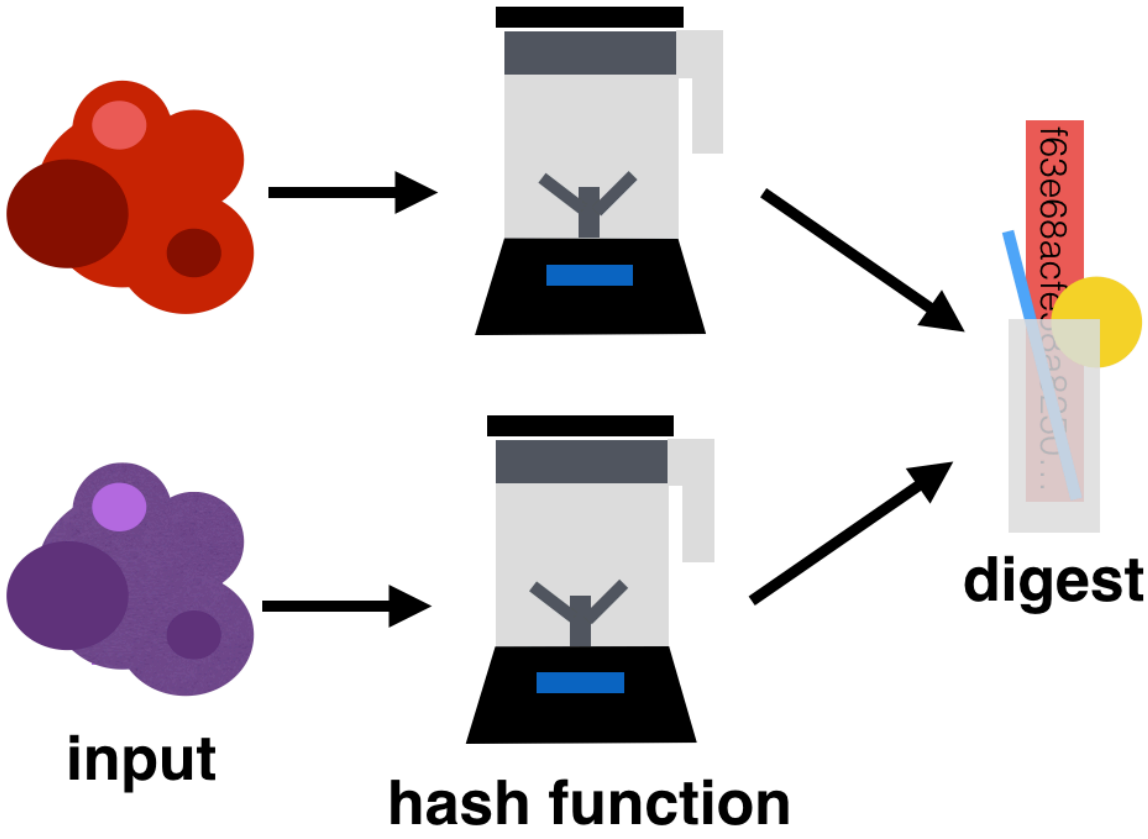


Figure 2.5 One should never be able to find two inputs (here represented on the left as two random blobs of data) that hash to the same output value (on the right). This security property is called collision resistance.

People often confuse collision resistance and second pre-image resistance. Take a moment to understand the differences.

NOTE

In addition, hash functions are usually designed so that their digests are unpredictable and random. This is useful because one cannot always prove a protocol to be secure thanks to one of the security properties of a hash function we've talked about (like collision resistance, for example). Many protocols are instead proven in the random oracle model where a fictional and ideal participant called a random oracle is used. In this type of protocol, one can send any inputs as requests to that random oracle which is said to return completely random outputs in response, and like a hash functions, giving it the same input twice will return the same output twice. Proofs in this model are sometimes controversial, as we don't know for sure if we can replace these random oracles with real hash functions in practice. Yet, many legimitate protocols are proven secure using this method where hash functions are seen as more ideal than they probably are.

2.3 Security considerations for hash functions

We've seen three security properties of a hash function:

- pre-image resistance
- second pre-image resistance
- collision-resistance

These security properties are often meaningless on their own, and it all depends on how you make use of the hash function. Nonetheless, it is important that we understand some limitations here before we look at some of the real-world hash functions.

First, these security properties assume that you are reasonably using the hash function. Imagine that I either hash the word "*yes*" or the word "*no*" and I then publish the digest. If you have some idea of what I was doing, you can simply hash both of the words and compare the result with what I gave you. Since there are no secrets involved, and since the hashing algorithm we've used is public, you are free to do that. And indeed, one could think this would break the pre-image resistance of the hash function, but we'll argue that your input was not "random" enough. Furthermore, since a hash function accepts an arbitrary-length input and always produce an output of the same length, there are also an infinite number of inputs that hash to the same output. Again, you could say "well isn't this breaking the second pre-image resistance?" Second pre-image resistance is merely saying that it is extremely hard to find another input, so hard we assume it's in practice impossible, but not theoretically impossible.

Second, the **size of the parameters** do matter. This is not a peculiarity of hash functions by any mean, all cryptographic algorithms must care about the size of their parameters in practice. Let's imagine the following extreme example, we have a hash function that produces outputs of length 2 bits in a uniformly random fashion (meaning that it will output `00` 25% of the time, `01` 25% of the time, and so on) You're not going to have to do too much work to produce a collision: after hashing a few random input strings you should be able to find two that hash to the same output. For this reason, there is a minimum output size that a hash function must produce in practice: 256 bits (or 32 bytes). With this large an output, collisions should be out of reach unless a breakthrough happens in computing.

How was this number obtained? In real-world cryptography, algorithms aim for a minimum of **128 bits of security**. It means that an attacker who wants to break an algorithm (providing 128-bit security) would have to perform around 2^{128} operations (for example, trying all the possible input strings of length 128-bit would take 2^{128} operations). For a hash function to provide all three security properties we mentioned earlier, it needs to provide at least 128 bits of security against all three attacks. The easiest attack is usually to find collisions, due to the **birthday bound**.

> **NOTE** The birthday bound takes its roots from probability theory, in which the birthday problem reveals some unintuitive results: how many people do you need in a room so that with at least 50% chance two people will share the same birthday (that's a collision). It turns out that 23 people taken at random are enough to reach these odds! Weird right? In practice, when we are randomly generating strings from a space of 2^N possibilities, you can expect someone to find a collision with 50% chance after having generated approximately $2^{N/2}$ strings.

If our hash function generates random outputs of 256 bits, the space of all outputs is of size 2^{256}. This mean that collisions can be found with good probability after generating 2^{128} digests (due to the birthday bound). This is the number we're aiming for, and this is why hash functions at a minimum must provide 256-bit outputs.

Certain constraints sometimes push developers to reduce the size of a digest by truncating it (removing some of its bytes). In theory this is possible, but can greatly reduce security. In order to achieve 128-bit security at a minimum, a digest must not be truncated under:

- 256-bit for collision-resistance.
- 128-bit for pre-image and second pre-image resistance.

This means that depending on what property one relies on, the output of a hash function could be truncated to obtain a shorter digest.

2.4 Hash functions in practice

As we've said earlier, in practice hash functions are rarely used alone. They are most often combined with other elements to either create a cryptographic primitive, or a cryptographic protocol. We will have many examples of using hash functions to build more complex objects in this book, but here are a few different ways hash functions have been used in the real-world:

Commitments. Imagine that you know that a stock in the market will increase in value and reach $50 in the coming month, but you really can't tell your friends about it (for some legal reason perhaps). You still want to be able to tell your friends that you knew about it, after the fact. Because you're smug (don't deny it). What you can do is to commit to a sentence like "*stock X will reach $50 next month*." To do this, hash the sentence and give your friends the output. A month later, reveal the sentence. Your friends will be able to hash the sentence to observe that indeed, it is producing the same output.

NOTE **Commitment schemes are generally trying to achieve two properties:**

- **Hiding. A commitment must hide the underlying value.**
- **Binding. A commitment must hide a single value. In other words, if you commit to a value x, you shouldn't be able to later successfuly reveal a different value y.**

Can you tell if a hash function provides both of these properties?

Subresource Integrity. It happens (often) that webpages import external JavaScript files. For example, a lot of websites will use Content Delivery Networks (CDNs) to import javascript libraries or web framework related files in their pages. These CDNs are placed in strategic locations in order to quickly deliver these files to visitors of the pages. Yet, if the CDN goes rogue, and decides to serve malicious javascript files, this could be a real issue. To counter this, webpages can use a feature called **subresource integrity** that allows the inclusion of a digest in the import tag:

```
<script src="https://code.jquery.com/jquery-2.1.4.min.js"
    integrity="sha256-8WqyJLuWKRBVhxXIL1jBDD7SDxU936oZkCnxQbWwJVw=">
  </script>
```

This is exactly the same scenario we talked about in the introduction of this chapter. Once the javascript file is retrieved, the browser will hash it (using SHA-256) and verify that it corresponds to the digest that was hardcoded in the page. If it checks out, the javascript file will get executed as its **integrity** has been verified.

BitTorrent. The BitTorrent protocol is used by users (called peers) all around the world to share files among each other directly (what we also call **peer-to-peer**). To distribute a file, it is cut into chunks and each chunk is individually hashed. These hashes are then shared as source of trust to represent the file to be downloaded. BitTorrent has several mechanisms to allow a peer to obtain the different chunks of a file from different peers. At the end, the integrity of the entire file is verified by hashing each of the downloaded chunks and matching the output to their respective known digests (before reassembling the file from the chunks). For example, the following "magnet link" represents The Ubuntu Operating System version 19.04. It is a digest (represented in hexadecimal) obtained from hashing metadata about the file, as well as all the chunks' digests.

```
magnet:?xt=urn:btih:b7b0fbab74a85d4ac170662c645982a862826455
```

Tor. The Tor browser's goal is to give individuals the ability to browse the internet anonymously. Another feature is that one can create hidden webpages, whose physical locations are difficult to track. Connections to these pages are secured via a protocol that uses the webpage's public key (we will see more about how that works in chapter 9 on session encryption). For example, *Silk Road* which used to be the eBay of drugs until it got seized by the

FBI, was accessible via `silkroad6ownowfk.onion` in the Tor browser. This base32 string actually represented the hash of Silk Road's public key. Thus by knowing the onion address, you can authenticate the public key of the hidden webpage you're visiting, and be sure that you're talking to the right page (and not an impersonator). If this is not clear, don't worry, I'll mention this again in chapter 9.

By the way, there is no way this string represents 256-bit (32-byte) right? How is this secure then according to what I said in section 2.3? Also, can you guess how *Dread Pirate Roberts* (the pseudonym of Silk Road's webmaster) managed to obtain a hash that contains the name of the website?

In all of these examples, a hash function was used to provide **content integrity** or **authenticity** in situations where:

- someone might tamper with the content being hashed
- the hash is **securely communicated** to you

We sometimes also say that we **authenticate** something or someone. It is important to understand that if the hash is not obtained securely, then anyone can replace it with the hash of something else, and thus it does not provide integrity by itself. The next chapter on message authentication code will fix this by introducing secrets. Let's now look at what actual hash function algorithms you can use.

2.5 Standardized hash functions

We've mentioned SHA-256 in our example above, which is only one of the hash functions one can use. Before we go ahead and list the recommended hash functions of our time, let's first mention other algorithms that people use in real-world applications that are not considered cryptographic hash functions.

First, functions like *CRC32* are **not** cryptographic hash functions but error-detecting code functions. While they will helpfully detect some simple errors they provide none of the above mentioned security properties and are not to be confused with the hash functions we are talking about here (even though they might share the name sometimes). Their output is usually referred to as checksum.

Second, popular hash functions like *MD5* and *SHA-1* are considered **broken** nowadays. While they both have been the standardized and widely accepted hash functions of the 90s, MD5 and SHA-1 were shown to be broken in 2004 and 2016 respectively when collisions were published by different research teams. These attacks were successful partly because of advances in computing, but mostly because flaws were found in the way the hash functions were designed.

NOTE

Both MD5 and SHA-1 were good hash functions until researchers demonstrated their lack of resistance from collisions. It remains that their (second) pre-image resistance has not been affected by these attacks. This does not matter for us, as we want to only talk about secure algorithms in this book. Nonetheless, you will sometimes still see people using MD5 and SHA-1 in systems that only rely on the pre-image resistance of these algorithms and not on their collision resistance. They will often argue that they cannot upgrade these hash functions to more secure ones because of legacy reasons. As the book is meant to last in time and be a beam of bright light for the future of real-world cryptography, this will be the last time we will mention these hash functions.

The next two sections will introduce SHA-2 and SHA-3 which are the two most widely used hash functions.

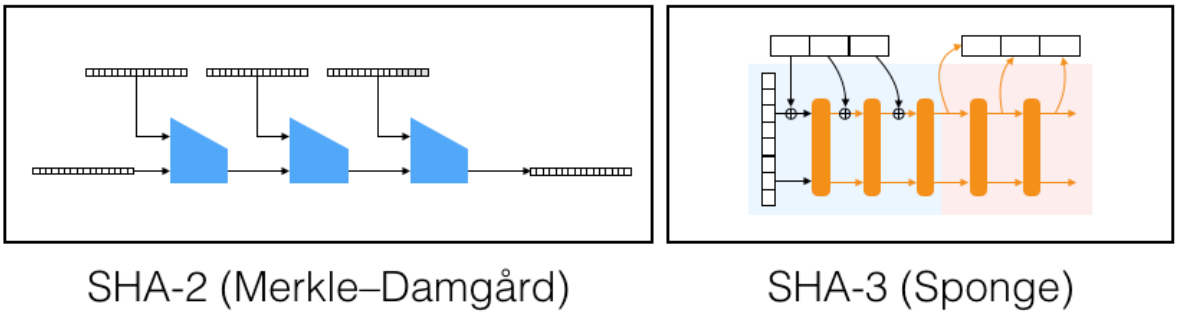


Figure 2.6 SHA-2 and SHA-3. The two most widely adopted hash functions. SHA-2 is based on the Merkle–Damgård construction, while SHA-3 is based on the sponge construction.

2.5.1 The SHA-2 hash function

Now that we have seen what hash functions are and had a glimpse at their potential use cases, it remains to be seen which hash functions one can use in practice. In the next two sections we introduce two widely accepted hash functions and we also give a high-level explanations of how they work from the inside. The latter part should not provide deeper insights on how to use hash functions, as the black box descriptions we gave should be enough, but nevertheless it is interesting to see how these cryptographic primitives were designed by cryptographers in the first place.

The most widely adopted hash function is the **Secure Hash Algorithm 2 (SHA-2)**. SHA-2 was invented by the NSA and standardized by the NIST in 2001. It was meant to add itself to the aging Secure Hash Algorithm 1 (SHA-1) already standardized by the NIST. SHA-2 provides 4 different versions producing outputs of 224, 256, 384 or 512 bits. Their respective names omit

the version of the algorithm: SHA-224, SHA-256, SHA-384, SHA-512. In addition, two other versions provide 224-bit and 256-bit by truncating the result of the larger versions SHA-512/224 and SHA-512/256 respectively.

Below we call each variant of SHA-2 with the OpenSSL CLI, observe that calling the different variants with the same input produce outputs of the specified lengths that are completely different.

```
$ echo -n "hello world" | openssl dgst -sha224
2f05477fc24bb4faefd86517156dafdecec45b8ad3cf2522a563582b
$ echo -n "hello world" | openssl dgst -sha256
b94d27b9934d3e08a52e52d7da7dabfac484efe37a5380ee9088f7ace2efcde9
$ echo -n "hello world" | openssl dgst -sha384
fdbd8e75a67f29f701a4e040385e2e23986303ea10239211af907fcbb83578b3e
417cb71ce646efd0819dd8c088de1bd
$ echo -n "hello world" | openssl dgst -sha512
309ecc489c12d6eb4cc40f50c902f2b4d0ed77ee511a7c7a9bcd3ca86d4cd86f9
89dd35bc5ff499670da34255b45b0cfd830e81f605dcf7dc5542e93ae9cd76f
```

Nowadays, people mostly use SHA-256 which provides the minimum 128 bits of security needed for our three security properties, while more paranoid applications make use of SHA-512.

Now, let's have a simplified explanation of how SHA-2 works.

It all starts with a special function called a **compression function**. A compression function takes two inputs of some size, and produces one output of the size of one of the inputs. Put simply: it takes some data, and returns less data. This is illustrated in figure 2.7.

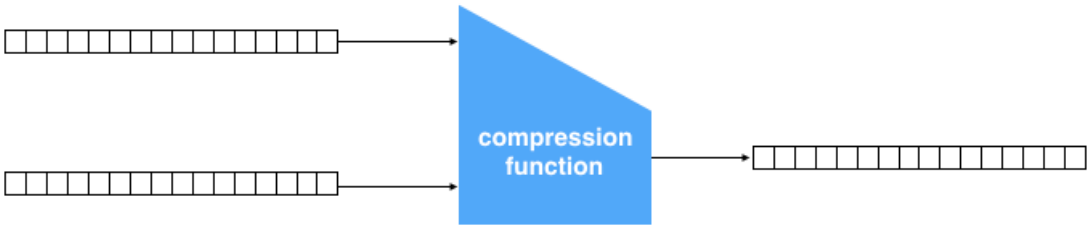


Figure 2.7 A compression function takes two different inputs of size X and Y (here both of 16 bytes) and returns an output of size either X or Y.

NOTE

While there are different ways of building a compression function, SHA-2 uses the Davies–Meyer method (see figure 2.8) which relies on a block cipher (a cipher that can encrypt a fixed-size block of data). I've mentioned the AES block cipher in chapter 1, but you haven't yet learned about them. For now accept the compression function as a blackbox until you read chapter 4 on authenticated encryption.

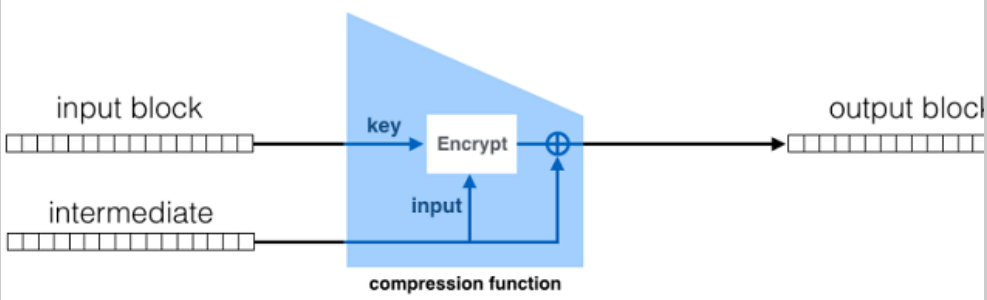


Figure 2.8 An illustration of a compression function built via the Davies-Meyer construc compression function's first input (the input block) is used as key to a block cipher. The sec o (the intermediate value) is used as input to be encrypted by the block cipher, it is then used XORing () itself with the output of the block cipher.

SHA-2 is a **Merkle–Damgård** construction, which is an algorithm (invented by Ralph Merkle and Ivan Damgård independently) that hashes a message by iteratively calling such a compression function. Specifically, it works by going through the following two steps:

1. It applies a **padding** to the input we want to hash, then **cuts the input into blocks** that can fit into the compression function. Padding means to append specific bytes to the input in order to make its length a multiple of some block size. Cutting the padded input into chunks of the same block size allows us to fit these in the first argument of the compression function. For example, SHA-256 has a block size of 512-bit. This step is illustrated in figure 2.9.

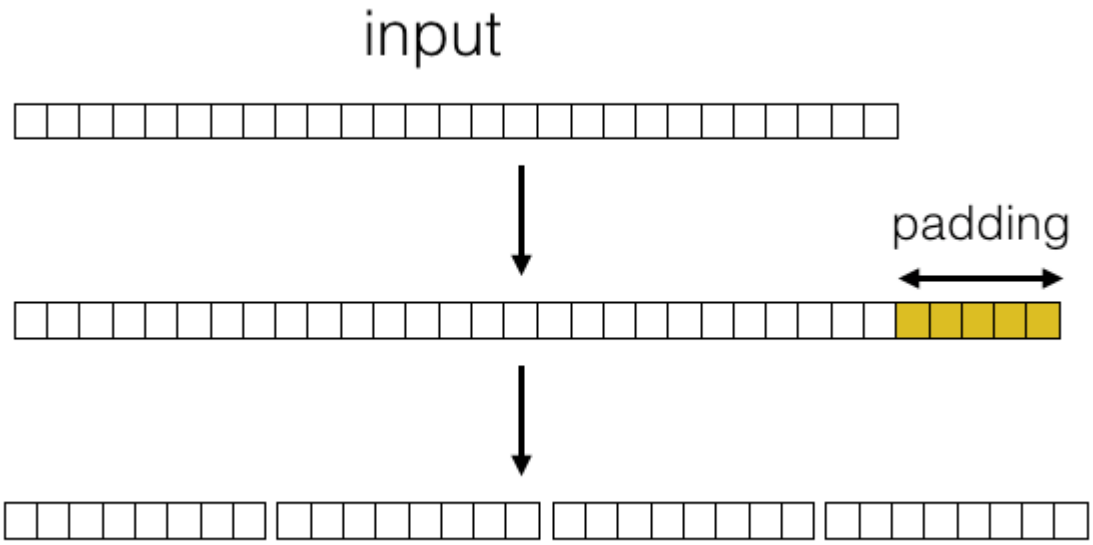


Figure 2.9 The first step of the Merkle–Damgård construction is to add some padding to the input message. After this step, the input should be of length a multiple of the input size of the compression function in use (for example 8 bytes). To do this we add 5 bytes of padding at the end to make it 32 bytes, we then cut the messages into 4 blocks of 8 bytes.

2. It **iteratively applies the compression function** to the message blocks, using the previous output of the compression function as second argument to the compression function. The final output is the digest. This is illustrated in figure 2.10.

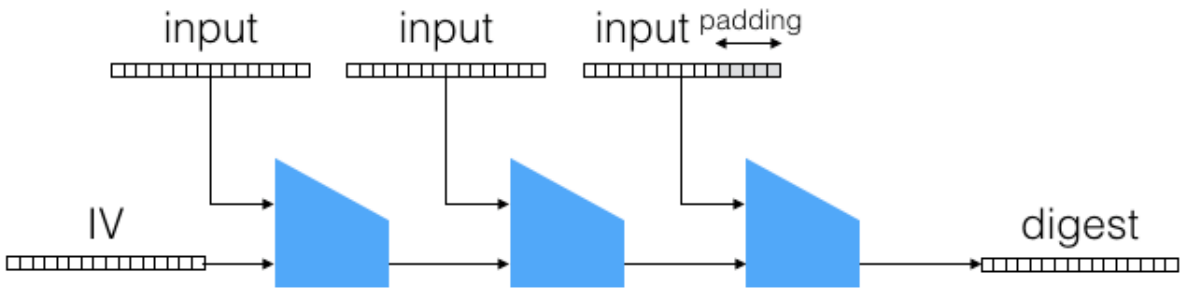


Figure 2.10 The Merkle–Damgård construction. It iteratively applies a compression function to each block of the input to be hashed, and the output of the previous compression function (except for the first step which takes an initial value `IV`). The final call to the compression function directly returns the digest.

And this is how SHA-2 works: by iteratively calling its compression function on fragments of the input until everything is processed into a final digest.

> **NOTE** The Merkle–Damgård construction has been proven to be collision resistant if the compression function itself is. Thus, the security of the arbitrary-length input hash function is reduced to the security of a fixed-sized compression function, which is easier to design and analyze. Therein lies the ingenuity of the Merkle–Damgård construction.

In the beginning, the second argument to the compression function is chosen to be a **nothing-up-my-sleeve** value. Specifically, SHA-256 uses the square roots of the first prime numbers to derive this value. A nothing-up-my-sleeve value is meant to convince the cryptographic community that it was not chosen to make the hash function weaker (for example, in order to create a backdoor). This is a popular concept in cryptography.

> **WARNING** Note that while SHA-2 is a perfectly fine hash function to use, it is not suitable for hashing secrets. This is because of a downside of the Merkle–Damgård construction which makes SHA-2 vulnerable to an attack called a length-extension attack if used to hash secrets. We will talk about this in more details in the next chapter.

2.5.2 The SHA-3 hash function

As I've mentioned earlier, both the MD5 and SHA-1 hash functions have been broken somewhat recently. These two functions made use of the same Merkle–Damgård construction I've described in the previous section. Because of this, and the fact that SHA-2 is vulnerable to length-extension attacks, the NIST decided in 2007 to organize an open competition for a new standard: **SHA-3**. This section will introduce the newer standard and will attempt to give a high-level explanation of its inner-workings.

In 2007, 64 different candidates from different international research teams entered the SHA-3 contest. 5 years later Keccak, one of the submissions, was nominated as the winner and took the name SHA-3. In 2015, SHA-3 was standardized in FIPS Publication 202.

SHA-3 observes the three previous security properties we've talked about and provides as much security as the SHA-2 variants. In addition, it is not vulnerable to length-extension attacks and can be used to hash secrets. For this reason, it is now the recommended hash function to use. It offers the same variants as SHA-2, this time indicating the full name SHA-3 in their named variants: SHA-3-224, SHA-3-256, SHA-3-384, and SHA-3-512. Thus, similarly to SHA-2, SHA-3-256 provides 256 bits of output, for example.

Let me now take a few pages to explain how SHA-3 works.

SHA-3 is a cryptographic algorithm built on top of a **permutation**. The easiest way to

understand a permutation is to imagine the following: you have a set of elements on the left, and the same set of elements on the right. Now trace arrows going from each element on the left to the right. Each element can only have one arrow starting from and terminating to them. You now have one permutation. This is illustrated in figure 2.11. By definition, any permutation is also **reversible**, meaning that from the output we can find the input back.

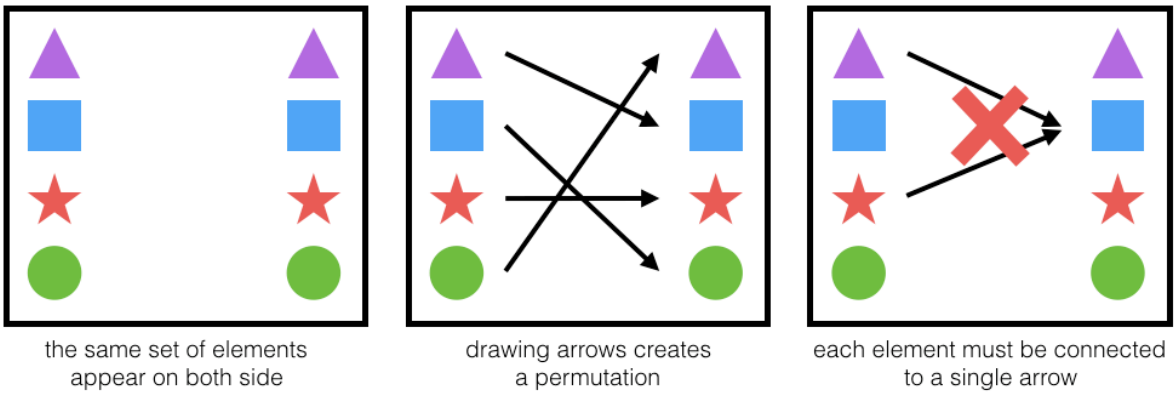


Figure 2.11 An example permutation acting on 4 different shapes. The permutation described by the arrows in the middle picture can be used to transform a given shape.

SHA-3 is built with a **sponge construction**, a different construction from Merkle–Damgård that was invented as part of the SHA-3 competition. It is based on a particular permutation called **keccak-f** that takes an input and returns an output of the same size.

> **NOTE** **We won't explain how keccak-f was designed, but you will get an idea in chapter 4 on authenticated encryption since it resembles the AES algorithm a lot (with the exception that it doesn't have a key). This is no accident, as one of the inventors of AES was also one of the inventors of SHA-3.**

In the next few pages I will use an 8-bit permutation to illustrate how the sponge construction works. Since the permutation is set in stone, you can imagine that figure 2.12 is a good illustration of the mapping created by this permutation on all possible 8-bit inputs. Compared to our previous explanation of a permutation, you can imagine that each possible 8-bit string is what we represented as different shapes (`000`… is a triangle, `100`… is a square, and so on).

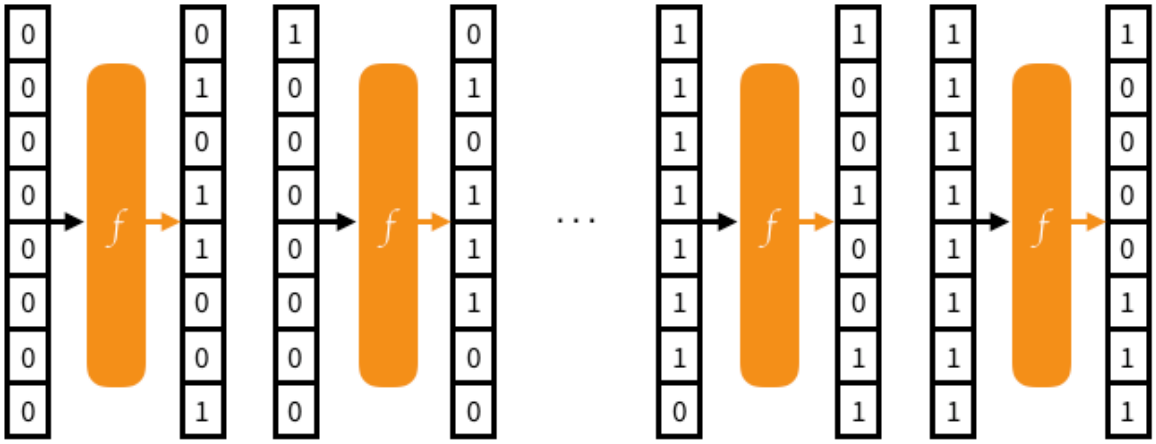


Figure 2.12 A sponge constructions makes use of a specified permutation `f`. By operating on an input, our example permutation creates a mapping between all possible input of 8 bits and all possible output of 8 bits.

To use a permutation in our sponge construction, we also need to define an arbitrary division of the input and the output into a **rate** and a **capacity**. It's a bit weird but stick with it. I illustrate this in figure 2.13.

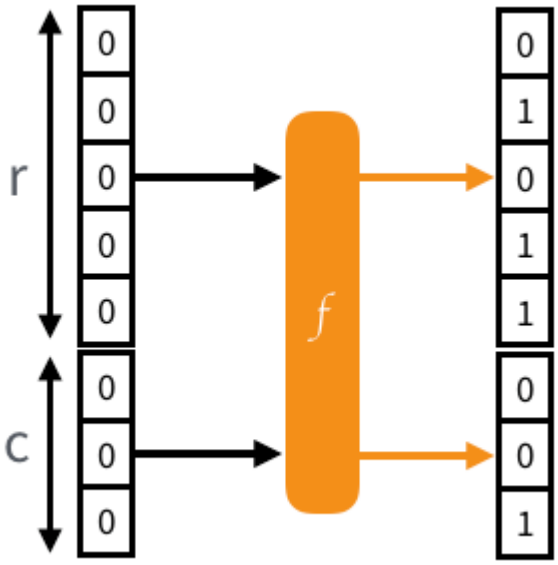


Figure 2.13 The permutation `f` randomizes an input of size 8 bits into an output of the same size. In a sponge construction, this permutation's input and output are divided into two parts: the rate (of size `r`) and the capacity (of size `c`).

Where we set the limit between the rate and the capacity is arbitrary. Different versions of SHA-3 will use different parameters. We informally point out that the capacity is to be treated like a secret, and the larger it is the more secure the sponge construction will be.

NOTE To understand what follows, you need to understand the XOR (Exclusive OR) operation. XOR is a bitwise operation, meaning that it operates on bits. I illustrate how it works in figure 2.14. XOR is ubiquitous in cryptography, so make you sure you remember it.

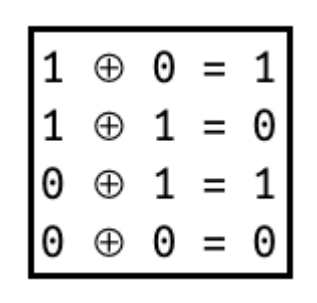


Figure 2.14 Exclusive OR or XOR (often denoted) operates on two bits. It is similar to the OR except for the case where both operands are 1s.

Now, like all good hash functions, we need to be able to hash something right? Otherwise it's a bit useless. To do that, we simply XOR () the input with the rate of the permutation's input. In the beginning, this is just a bunch of 0s. As we pointed out earlier, the capacity is to be treated like a secret so we won't XOR anything with it. I illustrate this in figure 2.15.

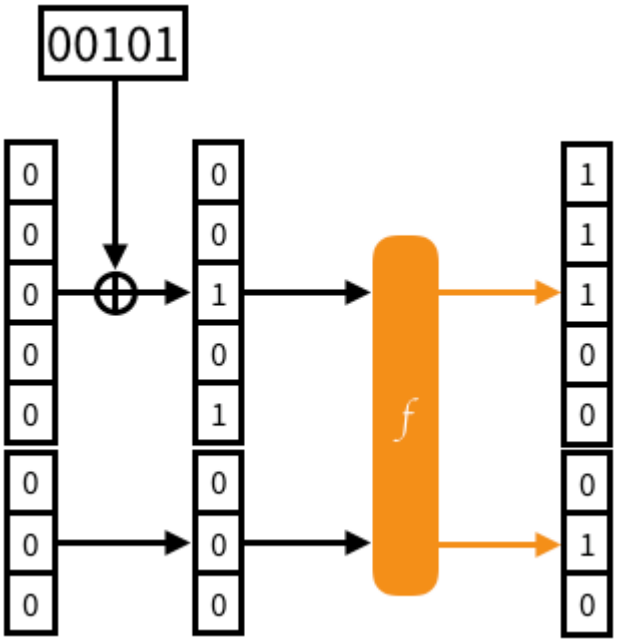


Figure 2.15 To absorb the 5 bits of input `00101`, a sponge construction with a rate of 5 bits can simply XOR the 5 bits with the rate (which is initialized to 0s). The permutation then randomizes the state.

The output obtained should now look random (although we can trivially find what the input is since a permutation is reversible by definition).

What if we want to ingest a larger input? Well, similarly to what we did with SHA-2:

1. pad your input if necessary (we will omit explanations of the padding), then divide your input into blocks of the rate size
2. iteratively call the permutation while XORing each block with the input of a permutation while permuting the **state** (the intermediate value output by the last operation) after each block has been XORed

I illustrate this in figure 2.16.

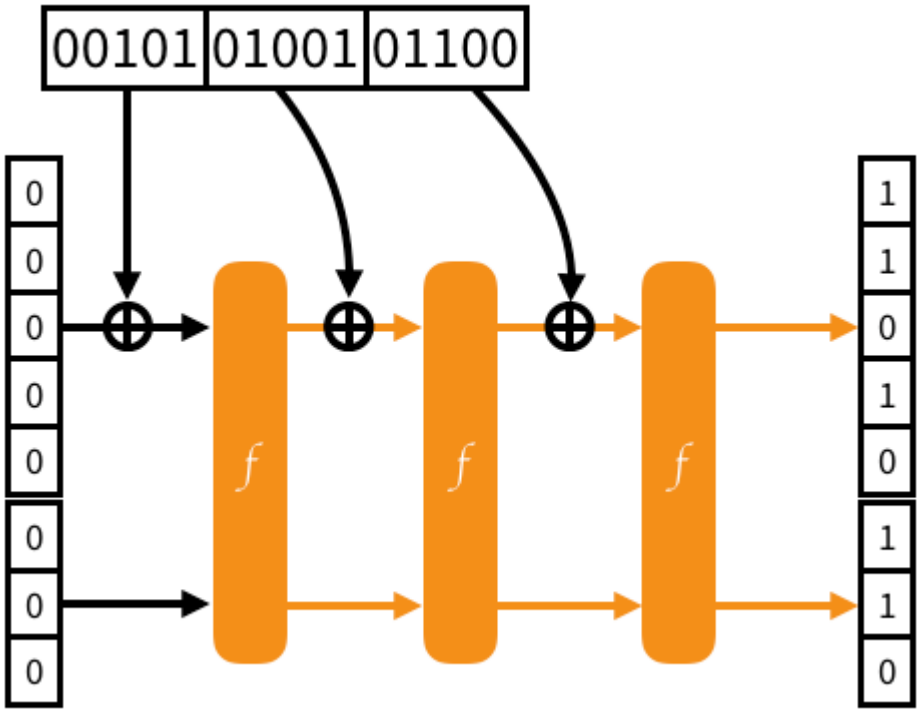


Figure 2.16 In order to absorb inputs larger than the rate size, a sponge construction will iteratively XOR input blocks with the rate and permutate the result.

So far so good, but we still haven't produced a digest. To do this, we can simply use the rate of the last state of the sponge (again, we are not touching the capacity). To obtain a longer digest, we can continue to permute and read from the rate part of the state as illustrated in figure 2.17.

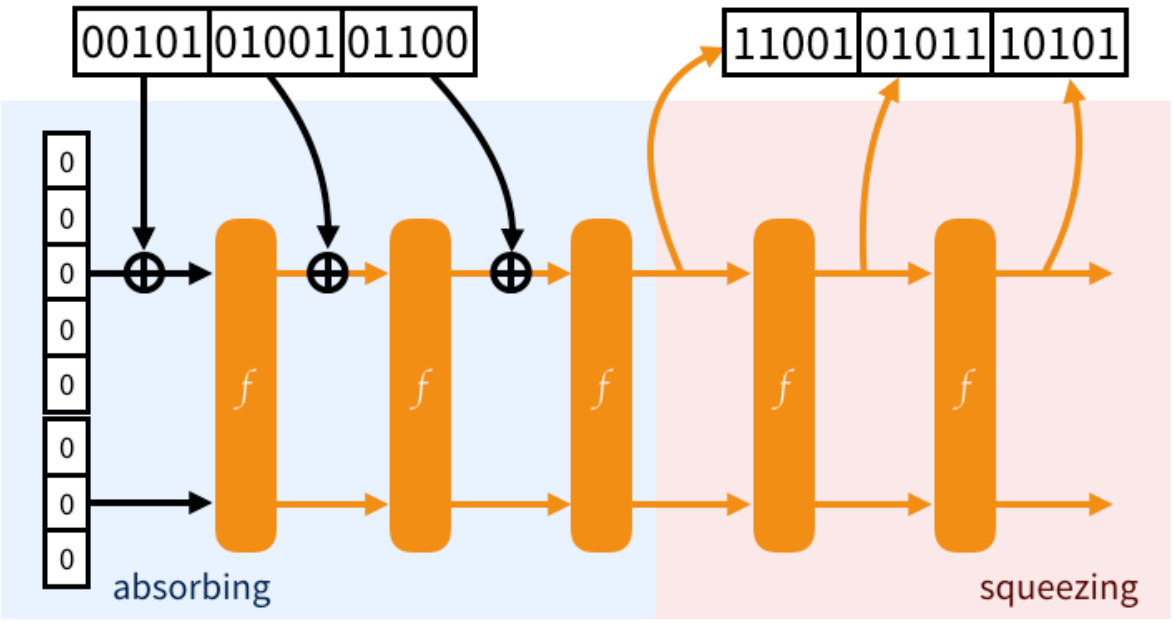


Figure 2.17 To obtain a digest with the sponge construction, one iteratively permutes the state and retrieves as much rate (the upper part of the state) as needed.

And this how SHA-3 works. Since it is a **sponge construction**, ingesting the input is naturally called *absorbing* and creating the digest is called *squeezing*. The sponge is specified with a 1600-bit permutation using different values for `r` and `c` depending on the security advertised by the different versions of SHA-3.

NOTE I've talked about random oracles earlier, an ideal and fictional construction that would return perfectly random response to queries and repeat itself if we query it with the same input. It turns out that the sponge construction has been proven to behave closely to a random oracle, as long as the permutation used by the construction looks random enough. How do we prove such security properties on the permutation? Our best approach is to try to break it, many times, until we get strong confidence in its design (which is what happened during the SHA-3 competition). The fact that SHA-3 can be modeled as a random oracle instantly gives it the security properties we would expect from a hash function.

2.5.3 SHAKE and cSHAKE, two extendable output functions (XOF)

I've introduced the two major hash function standards: SHA-2 and SHA-3. These are well defined hash functions that take arbitrary-length inputs and produce random-looking and fixed-length outputs. As you will see in later chapters, cryptographic protocols often necessitate this type of primitives but do not want to be constrained by the fixed sizes of a hash function's digest. For this reason, a more versatile primitive called an **extendable output function** or **XOF** (pronounced "zoff") was introduced by the SHA-3 standard. This section introduces the two standardized XoFs: **SHAKE** and **cSHAKE**.

SHAKE, specified in FIPS 202 along with SHA-3, can be seen as a hash function that can return an output of an arbitrary length. SHAKE is fundamentally the same construction as SHA-3, except that it is faster and permutes as much as you want it to permute in the squeezing phase. Producing outputs of different sizes is quite useful, not only to create a digest, but also to create random numbers, derive keys, and so on. I will talk about the different applications of SHAKE again in this book, for now just imagine that SHAKE is like SHA-3 except that it provides an output of any length you might want.

This construction is so useful in cryptography, that one year after SHA-3 was standardized, NIST published Special Publication 800-185 containing a **customizable SHAKE** called **cSHAKE**. cSHAKE is pretty much exactly like SHAKE, except that it also takes a **customization string**. This customization string can be empty, or can be any string you want. Let's first see an example of using cSHAKE in pseudo-code:

```
cSHAKE(input="hello world", output_length=256, custom_string="my_hash")
-> 72444fde79690f0cac19e866d7e6505c
cSHAKE(input="hello world", output_length=256, custom_string="your_hash")
-> 688a49e8c2a1e1ab4e78f887c1c73957
```

As you can see, the two digests differ, even though cSHAKE is as deterministic as SHAKE and SHA-3. This is because a different customization string was used. A customization string thus allows you to customize your XOF! This is useful in some protocols where, for example, different hash functions must be used in order to make a proof work. We call this **domain separation**. As a golden rule in cryptography: if the same cryptographic primitive is used in different use cases, do not use it with the same key (if it takes a key) or/and apply domain separation. You will see more example of domain separation as we survey cryptographic protocols in later chapters.

WARNING	**The NIST tends to specify algorithms that take parameters in bits instead of bytes. In the example above a length of 256 bits was requested. Imagine if you had requested a length of 16 bytes and got 2 bytes instead due to the program thinking you had requested 16 bits of output. This issue is sometimes called a bit attack.**

As with everything in cryptography the length of cryptographic strings like keys, parameters and outputs is strongly tied to the security of the system. It is important that one does not request too-short outputs from SHAKE or cSHAKE. **One can never go wrong by using an output of 256 bits**, as it will provide 128-bit of security against collision attacks, but real-world cryptography sometimes operates in constrained environments that could use shorter cryptographic values. This can be done if the security of the system is carefuly analyzed. For example, if collision resistance does not matter in the protocol making use of the value, pre-image resistance only needs 128-bit long outputs from SHAKE or cSHAKE.

2.5.4 Avoid ambiguous hashing with TupleHash

In this chapter, I have talked about different types of cryptographic primitives and cryptographic algorithms:

- The SHA-2 hash function, which is vulnerable to length-extension attacks but still widely used when no secrets are being hashed.
- The SHA-3 hash function, which is the recommended hash function nowadays.
- The SHAKE and cSHAKE XOFs, which are more versatile tools than hash functions as they offer a variable output length.

I will talk about one more handy function: **TupleHash**, that is based on cSHAKE and specified in the same standard as cSHAKE. TupleHash is an interesting function that allows one to hash a **tuple** (a list of something). To explain what TupleHash is and why it is useful, let me tell you a story.

A few years ago I was tasked to review a cryptocurrency as part of my work. It included basic features one would expect from a cryptocurrency: accounts, payments, and so on. Transactions between users would contain metadata about who is sending how much to who. It would also include a small fee to compensate the network for processing the transaction.

Alice, for example, can send transactions to the network, but to have them accepted she needs to include a proof that the transaction came from her. For this, she can hash the transaction, and sign it (I gave a similar example in the previous chapter). Anyone can hash the transaction and verify the signature on the hash to see that this is the transaction Alice meant to send. Figure 2.18 illustrates that a man-in-the-middle attacker who would intercept the transaction before it reached the network would not be able to tamper with the transaction. This is because the hash would change, and the signature would then not verify the new transaction digest.

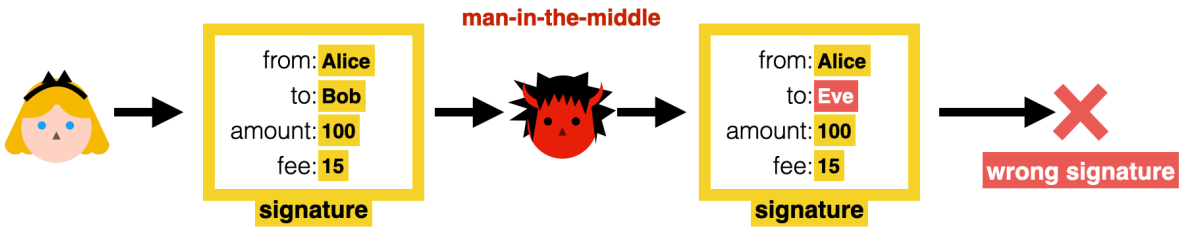


Figure 2.18 Alice sends a transaction, as well as a signature over the hash of the transaction. If a man-in-the-middle attacker attempts to tamper with the transaction, the hash will be different and thus the attached signature will be incorrect.

You will see in chapter 7 on signatures that such an attacker is, of course, unable to forge Alice's signature on a new digest. And thanks to the second pre-image resistance of the hash function used, the attacker cannot find a totally different transaction that would hash to the same digest either.

Is our man-in-the-middle attacker harmless? We're not out of the woods yet.

Unfortunately for the cryptocurrency I was auditing, the transaction was hashed by simply concatenating each field:

```
$ echo -n "Alice""Bob""100""15" | openssl dgst -sha3-256
34d6b397c7f2e8a303fc8e39d283771c0397dad74cef08376e27483efc29bb02
```

What can appear as totally fine actually completely broke the cryptocurrency's payment system. Doing this trivially allows an attacker to **break the second pre-image resistance** of the hash function.

Take a few moments to think about how you could find a different transaction that hashes to the same digest `34d6`....

What happens if we move one digit from the *fee* field to the *amount* field? One can see that the following transaction will hash to the same digest Alice signed:

```
$ echo -n "Alice""Bob""1001""5" | openssl dgst -sha3-256
34d6b397c7f2e8a303fc8e39d283771c0397dad74cef08376e27483efc29bb02
```

And thus, a man-in-the-middle attacker who would want Bob to receive a bit more money would be able to modify the transaction without invalidating the signature. As you've probably guessed, this is what **TupleHash** solves. It allows you to unambiguously hash a list of fields by using a non-ambiguous encoding. What happens in reality is something close to the following (with the `||` string concatenation operation):

```
cSHAKE(input="5"||"Alice"||"3"||"Bob"||"3"||"100"||"2"||"10",
output_length=256, custom_string="TupleHash"+"anything you want")
```

The input is this time constructed by prefixing each field of the transaction with its length. Take

a minute to understand why this solves our issue.

In general, one can use any hash function safely by always making sure to **serialize** the input before hashing it. Serializing the input means that there always exist a way to deserialize it (meaning to recover the original input). If one can deserialize the data, then there can't be any ambiguity on field delimitation.

2.6 Hashing passwords

You have seen several useful functions in this chapter that either are hash functions or extend hash functions. Before you can jump to the next chapter, I need to mention **password hashing**.

Imagine the following scenario: you have a website (which would make you a webmaster) and you want to have your users be able to register and login. So you create two web pages for these two respective features. Suddenly, you wonder: how are you going to store their passwords? Do you store them in clear in a database? There seems to be nothing wrong with this at first, you think. It is not perfect though, people tend to reuse the same password everywhere and if (or when) you get breached, and attackers manage to dump all of your users' passwords, it will be bad for them and it will be bad for the reputation of your platform. You think a little bit more, and you realize that an attacker who would be able to steal this database would then be able to log-in as any users. Storing the passwords in clear is now less than ideal and you would like to have a better way to deal with this.

One solution could be to hash your passwords and only store the digests. When someone logs in on your website, the flow would be the following:

- you receive the user's password
- you hash the password they give you and get rid of the password
- you compare the digest with what you had stored previously, if it matches the user is logged in

The flow allows you to handle their passwords for a limited amount of time. Still, an attacker that gets into your servers can stealthily remain to log passwords from this flow until you detect his presence. We acknowledge that this is still not a perfect situation but that we still improved its security. In security, we also call this **defense-in-depth**, which is the act of layering imperfect defenses in hope that an attacker will not defeat them all. This is what real-world cryptography is also about.

Other problems exist with this solution:

- If an attacker retrieves hashed passwords he can brute-force or do an exhaustive search (try all possible passwords) and test each attempt against the whole database. Ideally we would want an attacker to only be able to attack one hashed password at a time.
- Hash functions are supposed to be as fast as can be. Attackers can leverage this to

brute-force many many passwords per second. Ideally we would have a mechanism to slow down such attacks.

The first issue has been commonly solved by using **salts** which are random values that are public and different for each user. A salt is used along with the user's password when hashing it, which in some sense is like using a per-user customization string with cSHAKE: it effectively creates a different hash function for every user. Since each user uses a different hash function, an attacker cannot pre-compute large tables of passwords (called **rainbow tables**) in hope to test them against the whole database of stolen password hashes.

The second issue has been solved with **password hashes** which are designed to be slow. The current state of the art is **Argon2**, the winner of the Password Hashing Competition (password-hashing.net) that ran from 2013 to 2015. In practice, other non-standard algorithms are also used like PBKDF2, bcrypt and scrypt. The problem with these is that they can be used with insecure parameters and are thus not straight forward to configure in practice. In addition, only Argon2 and scrypt defend against heavy optimizations from attackers, as other schemes are not "memory-hard". memory-hard is a term used to mean that the algorithm can only be optimized through optimization of memory access. In other words, optimizing the rest doesn't gain you much. As optimizing memory access is limited, even with dedicated hardware (there's only so much cache you can put around a CPU), memory-hard functions will be slow to run on any type of device. This is a desired property when you want to prevent attackers from getting a non-negligible speed advantage in evaluating a function.

In figure 2.19, I recapitulate the different types of hash you've seen in this chapter.

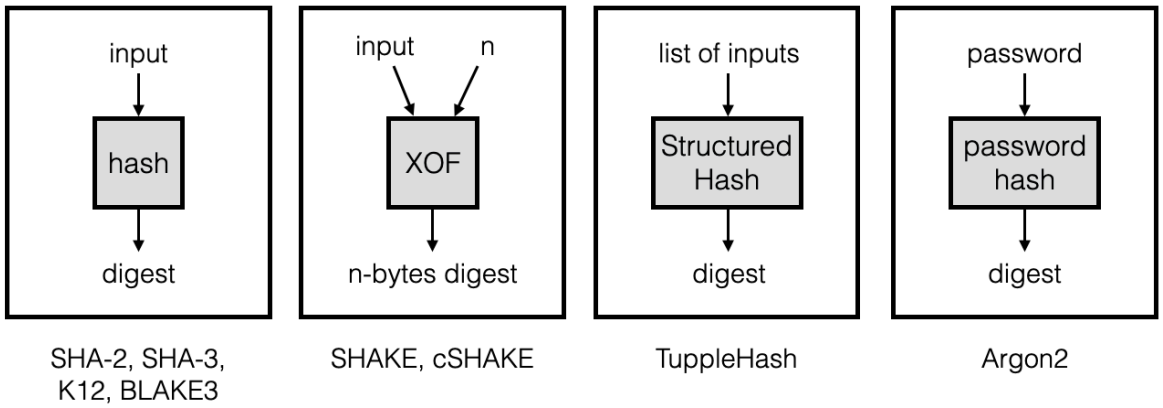


Figure 2.19 In this chapter you have seen four types of hash functions: the normal kind that provide a unique random-looking identifier for arbitrary-length inputs, extendable output functions which are similar but provide an arbitrary-length output, tuple hash functions that hash list of values unambiguously, and password hashing functions that can't be easily optimized in order to store passwords safely.

2.7 Summary

- A hash function provides collision resistance, pre-image resistance and second pre-image resistance.
  - Pre-image resistance means that one shouldn't be able to find the input that produced a digest.
  - Second pre-image resistance means that from an input and its digest, one shouldn't be able to find a different input that hashes to the same digest.
  - Collision resistance means that one shouldn't be able to find two random inputs that hash to the same output.
- The most widely adopted hash function is SHA-2, while the recommended hash function is SHA-3 due to SHA-2's lack of resistance to length-extension attacks.
- SHAKE is an extendable output function (XOF) that acts like a hash function but provides an arbitrary-length digest.
- cSHAKE for customizable SHAKE allows one to easily create instances of SHAKE that behave like different XoFs. This is called domain separation.
- Objects should be serialized before being hashed, in order to avoid breaking the second pre-image resistance of the hash function. Algorithms like TupleHash automatically take care of this.
- Hashing passwords make use of slower hash functions designed specifically for that purpose. The state of the art being Argon2.

3 Message authentication codes

This chapter covers

- Message authentication codes (MACs), a cryptographic primitive to protect the integrity of data.
- The security properties and the pitfalls of MACs.
- The widely adopted standards for MACs.

In chapter 2, you've learned about an interesting construction --hash functions—that on its own doesn't provide much, but if used in combination with a secure channel allows you to verify the authenticity and integrity of some data. In this chapter, you will see how you can provide integrity and authenticity over messages without the use of a secure channel at all.

For this chapter you'll need to have read:

- Chapter 2 on hash functions.

3.1 Stateless cookies, a motivating example for message authentication codes

Let's picture the following scenario: you are a webpage. You're bright, full of colors, and above all you're proud of serving a community of loyal users. To interact with you, visitors must first log-in by sending you their credentials, which you must then validate. If the credentials match those that were used when the user first signed up, then you have successfully **authenticated** the user.

Of course, a web browsing experience is composed not just of one, but of many requests. To avoid having the user re-authenticate in every request, you can make their browser store the user

credentials and re-send them automatically within each request. Browsers have a feature just for that: **cookies**! Cookies are not just for credentials, they can store anything you'd want the user to send you within each of their requests.

While this naive approach works well, usually you don't want to store sensitive information like user passwords in clear in the browser. Instead, a session cookie most often carries a **random string**, generated right after a user has logged-in. The webserver stores the random string in a temporary database under a user's nickname. If the browser publishes the session cookie somehow, no information about the user's password is leaked (although it can be used to impersonate the user). The webserver also has the possibility to kill the session by deleting the cookie on their side, which is nice.

There is nothing wrong with this approach, but in some cases it might not scale well. If you have many servers, it could be annoying to have all the servers share the association between your users and the random strings. Instead, you could store more information on the browser side. Let's see how we could do this.

Naively, you could have the cookie contain a username instead of a random string, but this is obviously an issue as I can now impersonate any user by manually modifying the username contained in my cookie. Perhaps the hash functions you learned about in chapter 2 can help us… Take a few minutes to think of a way they could prevent a user from tampering with their own cookies.

A second naive approach could be to store not only a username, but a digest of that username as well in a cookie. You could use a hash function like SHA-3 to hash the username. I illustrate this in figure 3.1. Do you think this can work?

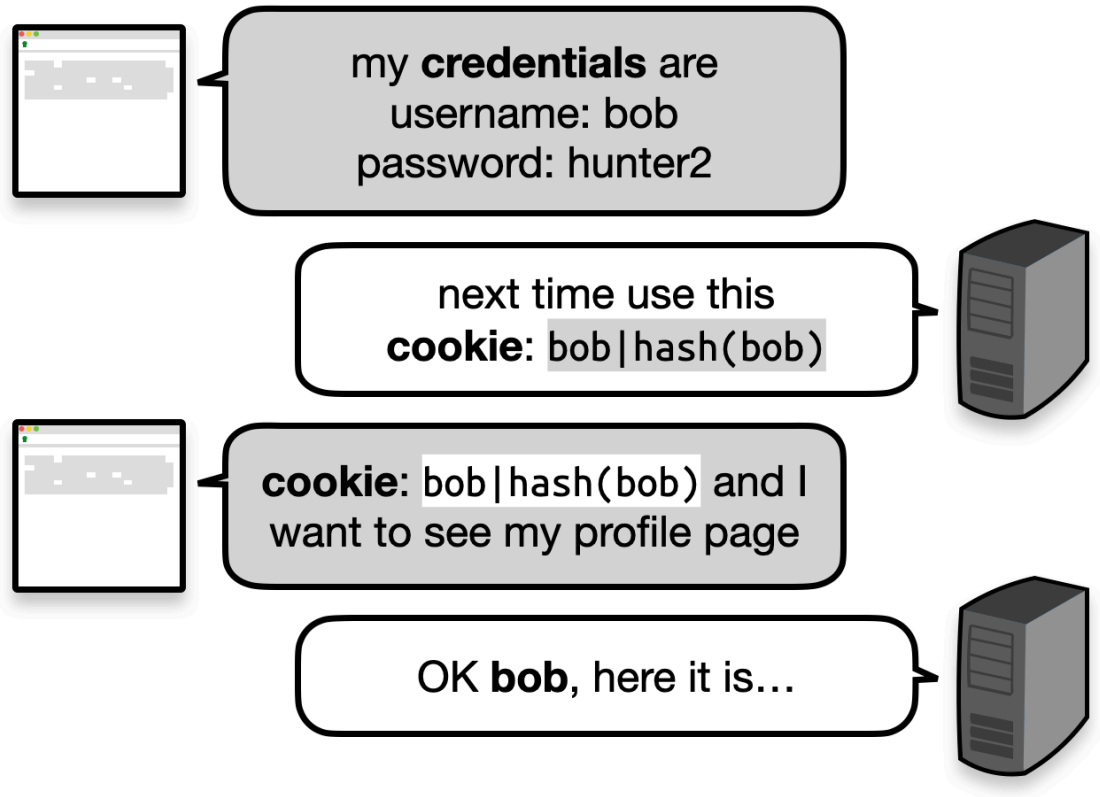


Figure 3.1 To authenticate the requests of a browser, a webserver asks the browser to store a username and a hash of that username, and to send this information in every subsequent requests.

There's a big problem with this approach. Remember, the hash function you are using is a public algorithm and can be recomputed on new data by a malicious user. If you do not trust the origin of a hash, it does not provide data integrity! Indeed, figure 3.2 shows that if a malicious user modifies the username in their cookie, they can also simply recompute the digest part of the cookie.

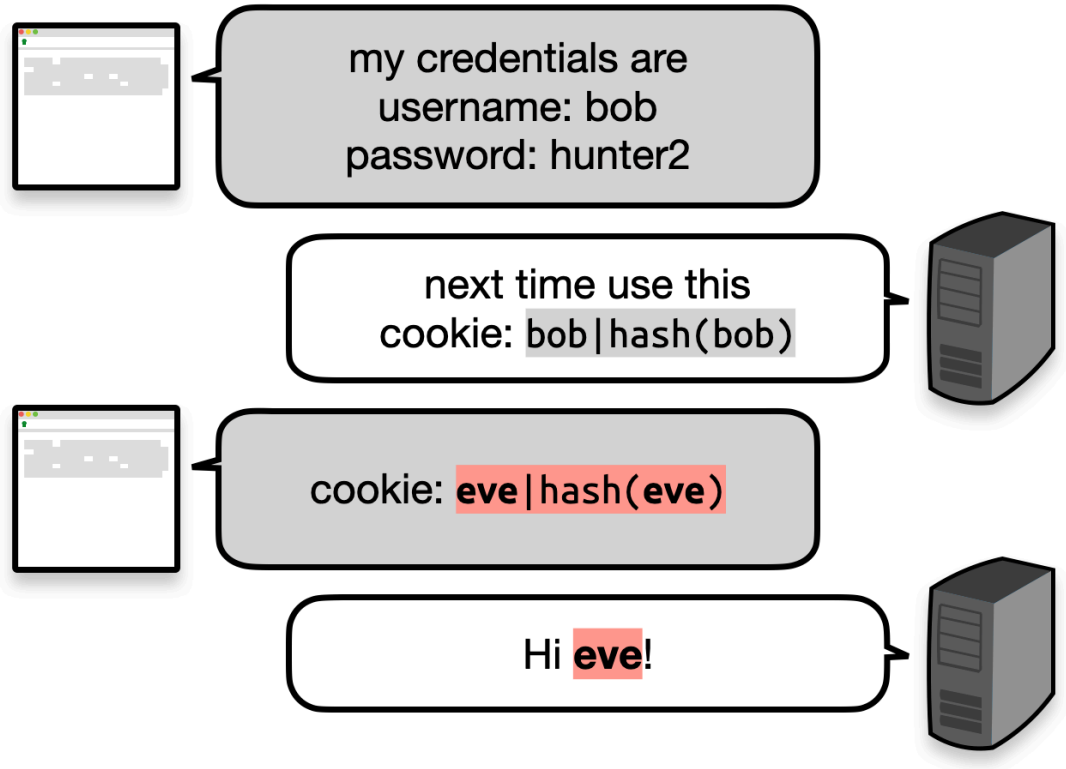


Figure 3.2 A malicious user can modify the information contained in their cookies. If a cookie contains a username and a hash, both can be modified to impersonate a different user.

Still, using a hash is not a stupid idea. What else can we do? Turns out that there is a similar primitive to the hash function called a **message authentication code** (**MAC**) that will do exactly what we need. (Do not focus on the name too much, as it has poorly aged.)

A MAC is a secret-key algorithm that takes an input, like a hash function, but also a secret key. It then produces a unique output called an authentication tag. This process is **deterministic**: given the same secret key and the same message, the same authentication tag will be produced.

I illustrate this in figure 3.3.

Figure 3.3 The interface of a message authentication code (MAC). The algorithm takes a secret key and a message, and deterministically produces a unique authentication tag. Without the key, it should be impossible to reproduce that authentication tag.

To make sure a user can't tamper with their cookie, let's now make use of this new primitive. When the user logs in for the first time: produce an authentication tag from your secret key and their username, and have them store their username and the authentication tag in a cookie. Since they don't know the secret key, they won't be able to forge a valid authentication tag for a different username.

To validate their cookie, do the same: produce an authentication tag from your secret key and the username contained in the cookie, and check if it matches the authentication tag contained in the cookie. If it matches, it must have come from you, as you were the only one who could have produced a valid authentication tag (under your secret key). I illustrate this in figure 3.4.

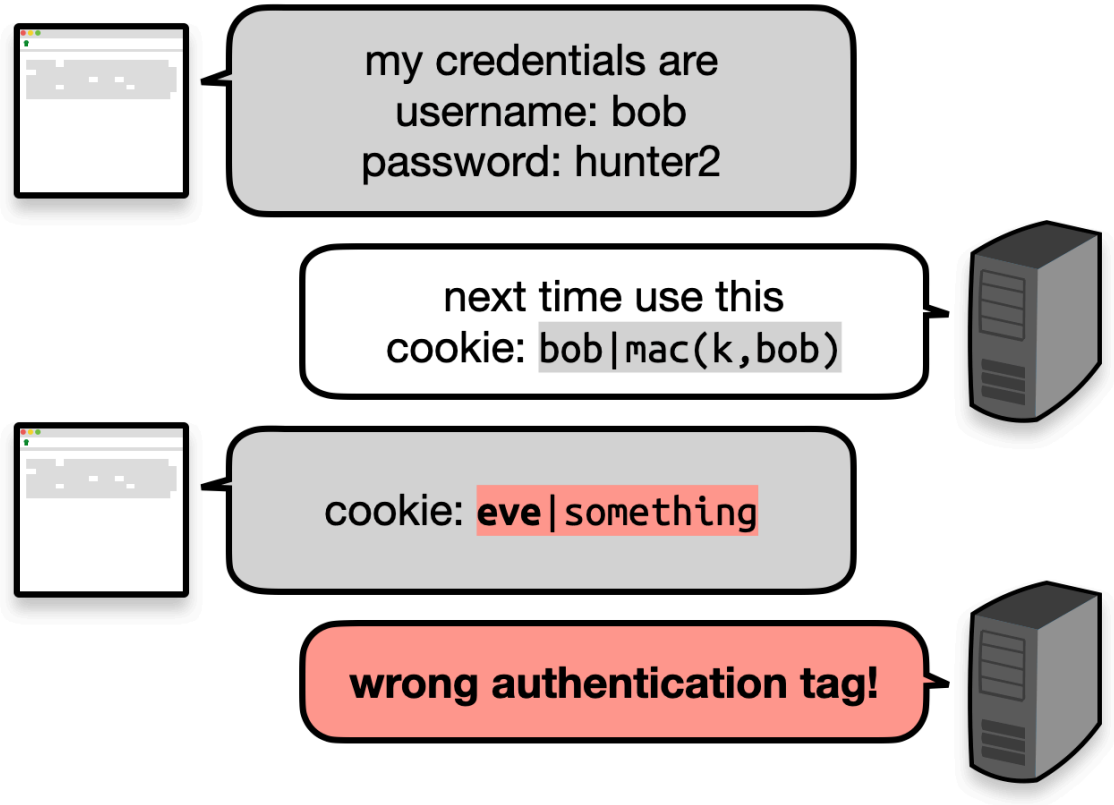


Figure 3.4 A malicious user tampers with his cookie, but cannot forge a valid authentication tag for the new cookie. Subsequently, the webpage cannot verify the authenticity and integrity of the cookie, and thus discard the request.

A MAC is like a private hash function, that only you can compute (because you know the key). In a sense, you can personalize a hash function with a key. The relation with hash functions doesn't stop there, as you will see later in this chapter that MACs are often built from hash functions.

Next, let's see a different example using real code.

3.2 An example in code

So far you were the only one using a MAC. Let's increase the number of participants, and let's use that as a motivation to write some code to see how MACs are used in practice.

Imagine that you want to communicate with someone else, and you do not care about other people reading your messages. What you really care about though, is the integrity of the messages: they must not be modified. A solution is to have both you and your correspondent use the same secret key with a MAC to protect the integrity of your communications.

For this example, we'll use one of the most popular MAC functions: **Hash-Based Message Authentication Code (HMAC)** with the **Rust** programming language. HMAC is a MAC that uses a hash function at its core. It is compatible with different hash functions, but it is usually mostly used in conjunction with SHA-2.

The sending part simply takes a key and a message, and returns an authentication tag.

Listing 3.1 sending.rs

```
use sha2::Sha256;
use hmac::{Hmac, Mac, NewMac};

fn send_message(key: &[u8], message: &[u8]) -> Vec<u8> {
    let mut mac = Hmac::<Sha256>::new(key.into());   ❶

    mac.update(message);   ❷

    mac.finalize().into_bytes().to_vec()   ❸
}
```

❶ We instantiate HMAC with a secret key and the SHA-256 hash function.

❷ We buffer more input for HMAC.

❸ A call to `finalize` returns the authentication tag.

On the other side, the process is similar. After receiving both the message and the authentication tag, your friend can generate their own tag with the same secret key, and compare them. Similarly to encryption, both sides need to share the same secret key to make this work.

Listing 3.2 receiving.rs

```
use sha2::Sha256;
use hmac::{Hmac, Mac, NewMac};

fn receive_message(key: &[u8], message: &[u8],
  authentication_tag: &[u8]) -> bool {
    let mut mac = Hmac::<Sha256>::new(key);   ❶
    mac.update(message);   ❷

    mac.verify(&authentication_tag).is_ok()
}
```

❶ The receiver needs to recreates the authentication tag from the same key and message.

❷ A call to `verify` checks if the reproduced authentication tag matches the received one.

Note that this protocol is not perfect: it allows replays. If a message and its authentication tag are replayed at a later point in time, they'll still be authentic, but you'll have no way of detecting that it is an older message being resent to you. Later in this chapter I'll tell you about a solution.

Now that you know what a MAC can be used for, I'll talk about some of the gotchas of MACs in the next section. It should answer your questions about the `counter` we used, and why we cannot directly compare authentication tags (which is a common mistake).

3.3 Security properties of a message authentication code

MACs, like all cryptographic primitives, have their oddities and pitfalls. Before going any further, I will provide a few explanations on what security properties MACs provide and how to use them correctly. You will learn in this order:

- MACs are resistant against forgery of authentication tags.
- An authentication tag needs to be of a minimum length to be secure.
- Messages can be replayed if authenticated naively.
- Verifying an authentication tag is prone to bugs.

Let's get started!

3.3.1 Forgery of authentication tag

The general security goal of a MAC is to prevent **authentication tag forgery** on a new message. This means that without knowledge of the secret key `k`, one cannot compute the authentication tag `t = MAC(k, m)` on messages `m` of their choice. This sounds fair right? We can't compute a function if we're missing an argument.

MACs provide much more assurance than that. Real-world applications often let attackers obtain authentication tags on some constrained messages. For example, this was the case in our previous introduction scenario: a user could obtain almost-arbitrary authentication tags by registering with an available nickname. Hence, MACs have to be secure even against these more powerful attackers. A MAC usually comes with a proof that even if an attacker can ask you to produce the authentication tags for a large number of arbitrary messages, the attacker should still not be able to forge an authentication tag on a never-seen-before message by themselves.

NOTE **One could wonder how proving such an extrem property is useful. If the attacker can directly request authentication tags on arbitrary messages then what is there left to protect? But this is how security proofs work in cryptography: they take the most powerful attacker and show that even then, the attacker is hopeless. In practice, the attacker is usually less powerful and thus we have confidence that, if a powerful attacker can't do something bad, a less powerful one has even less recourse.**

As such, you should be protected against such forgeries, **as long as the secret key used with the MAC stays secret**. This implies that the secret key has to be random enough (more on that in chapter 8) and large enough (usually 16 bytes). Furthermore, a MAC is vulnerable to the same type of **ambiguous attack** we've seen in chapter 2. If you are trying to authenticate structures, make sure to serialize them before authenticating them with a MAC, otherwise forgery might be trivial.

3.3.2 Lengths of authentication tag

Another possible attack against usage of MACs are **collisions**.

Remember, finding a collision for a hash function means finding two different inputs `X` and `Y` such that `HASH(X) = HASH(Y)`. We can extend this definition to MACs by defining a collision when `MAC(k, X) = MAC(k, Y)` for inputs `X` and `Y`. As we've learned in the previous chapter 2 with the **birthday bound**, collisions can be found with high probability if the output length of our algorithm is too small. For example with MACs, an attacker who has access to a service producing 64-bit authentication tags can find a collision with high probability by requesting a much lower number (2^{32}) of tags. Such a collision is rarely exploitable in practice, but there exist some scenarios where collision resistance matters. For this reason, we would want an authentication tag size that would limit such attacks. In general, 128-bit authentication tags are used as they provide enough resistance.

> *[requesting 2^{64} authentication tags] would take 250,000 years in a continuous 1Gbps link, and without changing the secret key K during all this time*
>
> *– RFC 2104 HMAC: Keyed-Hashing for Message Authentication*

Using 128-bit authentication tag might appear counter-intuitive, since we wanted 256-bit outputs for hash functions. But hash functions are public algorithms that one can compute **offline**, which allows an attacker to optimize and parallelize an attack heavily. With a keyed function like a MAC, an attacker cannot efficiently optimize the attack offline and is forced to directly request authentication tags from you (which usually makes the attack much slower). A 128-bit authentication tag would require 2^{64} **online** queries from the attacker in order to have a 50% chance to find collisions, which is deemed large enough. Nonetheless, one might still want to increase an authentication tag to 256-bit, which is possible as well.

3.3.3 Replay attacks

One thing I still haven't mentioned are **replay attacks**. Imagine that Alice and Bob communicate in the open (using an insecure connection) and append each of their messages with an authentication tag in order to protect them from tampering. To do that they both use two different secret keys, `k1` for protecting the integrity of messages coming from Bob and `k2` for protecting the integrity of messages coming from Alice (we've mentioned this concept called **domain separation** in the previous chapter 2. To avoid ambiguity you should always use different keys or parameters for different use cases). See figure 3.5.

Figure 3.5 Two users sharing two keys `k1` and `k2` exchange messages, along with authentication tags. These tags are computed out of `k1` or `k2` depending on the direction of the messages. A malicious observer replays one of the messages to the user.

In this scenario, nothing prevents a malicious observer from replaying one of the messages to its recipient. A protocol relying on MAC must be aware of this and build protections against this. One way is to add an incrementing counter to the input of the MAC as in figure 3.6.

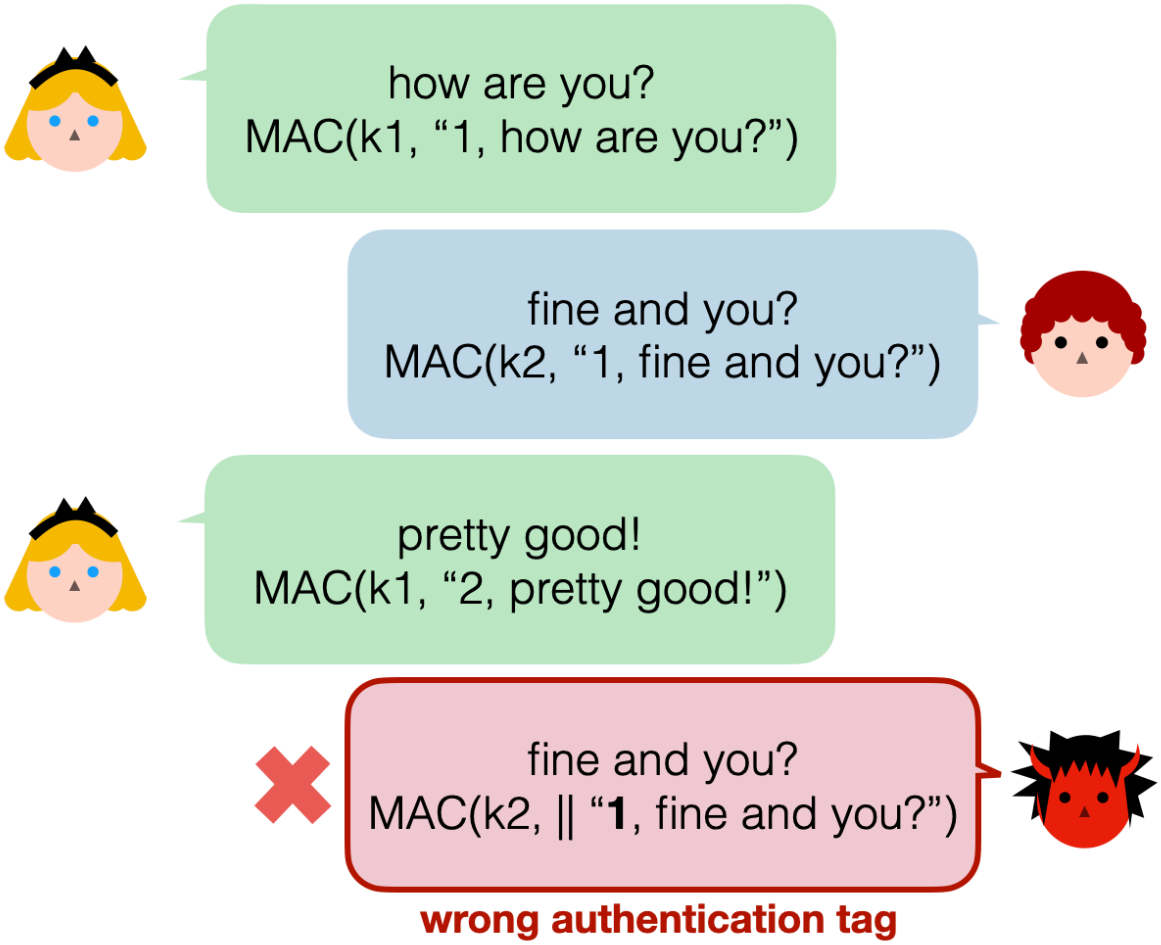


Figure 3.6 Two users sharing two keys `k1` and `k2` exchange messages, along with authentication tags. These tags are computed out of `k1` or `k2` depending on the direction of the messages. A malicious observer replays one of the messages to the user. Since the victim has incremented his counter, the tag will be computed over `2, fine and you?` and will not match the tag sent by the attacker. This will allow the victim to successfully reject that replayed message.

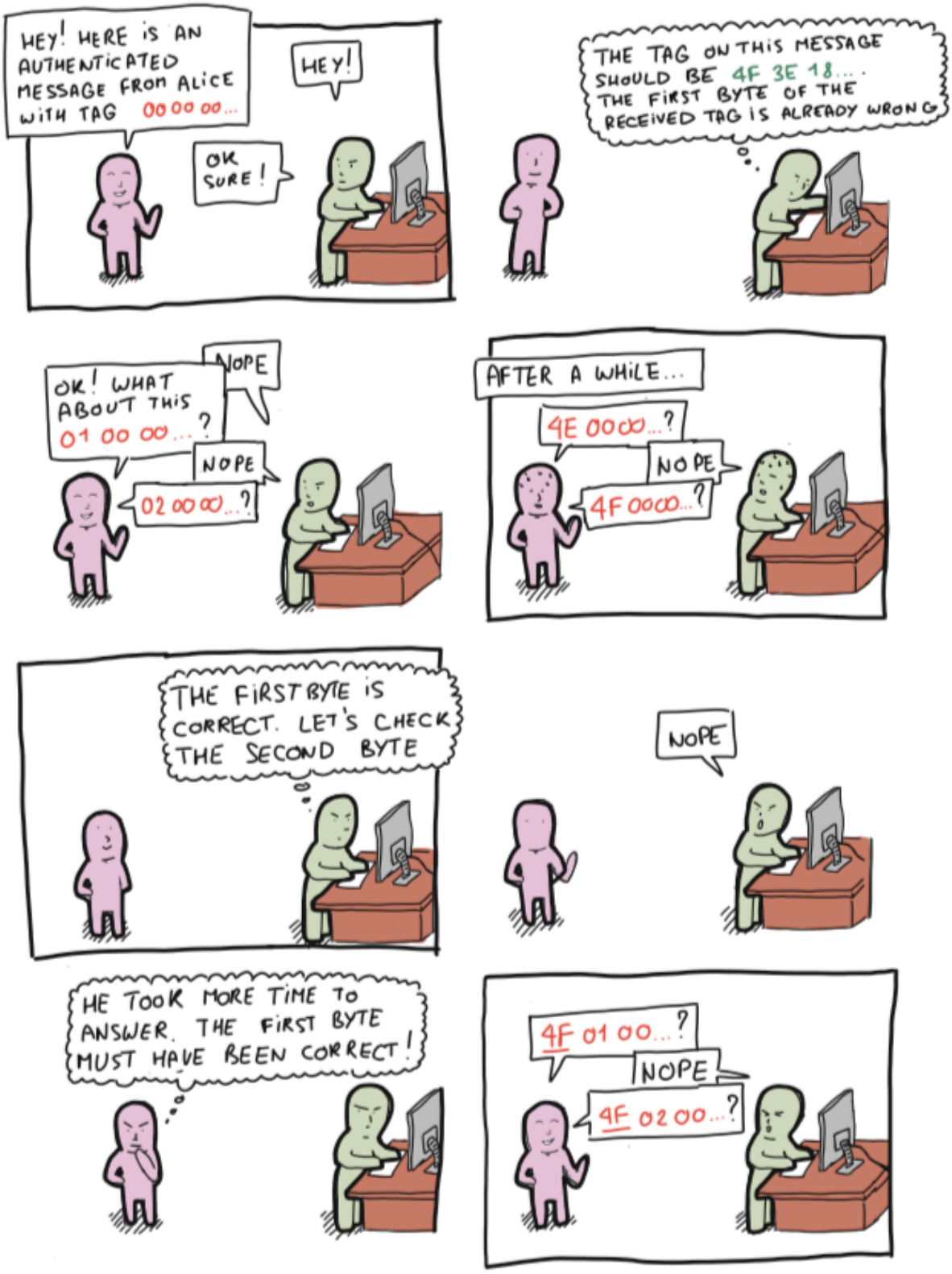
In practice, counters are often a fixed 64-bit length. This allows one to send 2^{64} messages before filling up the counter (and risking it to wrap around and repeat itself). Of course, if the shared secret is **rotated** frequently (meaning that after X messages, participants agree to use a new shared secret), then the size of the counter can be reduced and reset to `0` after a key rotation. (You should convince yourself that reusing the same counter with two different keys is OK.) Counters are **never variable-length** because of ambiguous attacks (again). Can you figure out how a variable-length counter could possibly allow an attacker to forge an authentication tag? You can find an example in the answer appendix at the end of this book.

3.3.4 Verifying authentication tags in constant-time

This last one is dear to me, as I found this vulnerability many times in different applications I’ve audited as part of my work.

NOTE **As you will see, this vulnerability is quite subtle. It belongs to the larger class of attacks called side-channel attacks, as it does not exploit the algorithm directly, but indirectly by analyzing some information leaked by the algorithm being run. Side-channel attacks can be exploited by measuring the power consumption of a device during a cryptographic operation, or the electromagnetic radiations emitted, or even by analyzing the time an algorithm takes to run (which is the case here).**

When verifying an authentication tag, the comparison between the received authentication tag and the one you computed must be done in **constant-time**. This means the comparison should always take the same time (assuming the received one is of the correct size). If the time it takes to compare the two authentication tags is not constant-time, it is probably because it returns the moment the two tags differ. This usually gives enough information to enable attacks that can recreate a valid authentication tag byte-by-byte by measuring how long it took for the verification to finish. I explain this in the following comicstrip. We call these types of attacks **timing attacks**.



Fortunately for us, cryptographic libraries implementing MACs also provide convenient functions to verify an authentication tag in constant-time. If you're wondering how this is done, here is how Golang implements an authentication tag comparison in constant-time code:

Listing 3.3 constant_time.go.go

```
for i := 0; i < len(x); i++ {
    v |= x[i] ^ y[i]
}
```

The trick is that no branch is ever taken. How this works exactyl is left as an exercise for the reader.

3.4 MAC in the real-world

Now that I have introduced what MACs are and what security properties they provide, let's take a look at how people use them in real settings.

Authentication of Messages. MACs are used in many places to ensure that the communication between two machines or two users was not tampered with. This is necessary in both cases where communications are in clear and where communications are encrypted. I have already explained how this happens when communications are transmitted in clear, and I will explain how this is done when communications are encrypted in the next chapter 4 on Authenticated Encryption.

Deriving keys. One particularity of MACs is that they are often designed to produce bytes that look random (like hash functions). This property can be utilized to use a single key to generate random numbers or to produce more keys. In chapter 8 on secrets and randomness, I will introduce the **HMAC-based Key Derivation Function (HKDF)** that does exactly this by using **HMAC**, one of the MAC algorithms we will talk about in this chapter.

> **NOTE** Imagine the set of all functions that take a variable-length input and produce a random output of a fixed-size. If we could pick a function at random from this set, and use it as a MAC (without a key), it would be swell. We would just have to agree on which function (kind of like agreeing on a key). Unfortunately, we can't have such a set (it is way too large) and thus we can emulate picking such a random function by designing something close enough: we call such constructions pseudo-random function (PRF). HMAC and most practical MACs are such constructions: they are randomized by the key argument instead. Choosing a different key, is like picking a random function. Caution though, as not all MACs are PRFs.

Integrity of Cookies. To track your users' browser sessions you can send them a random string (associated to their metadata) or send them the metadata directly, attached with an authentication tag so that they cannot modify it. This is what I explained in the introduction example.

Hash Tables. Programming languages usually expose data structures called hash tables (also

called hashmaps, dictionaries, associated arrays, and so on) that make use of non-cryptographic hash functions. If a service exposes this data structure in such a way where the input of the non-cryptographic hash function can be controlled by attackers, this can lead to **denial of service** (DoS) attacks (meaning that an attacker can render the service unusable). To avoid this, the non-cryptographic hash function is usually randomized at the start of the program. Many major applications have decided to use a MAC with a random key in place of the non-cryptographic hash function. This is the case for many programming languages like Rust, Python, Ruby or major applications like the Linux kernel that all make use of SipHash (a poorly-named MAC optimized for short authentication tags) with a random key generated at the start of the program.

3.5 Message authentication codes in practice

You've learned that MACs are cryptographic algorithms that can be used between one or more parties in order to protect the integrity and the authenticity of information. As widely-used MACs also exhibit good randomness, MACs are also often used to produce random numbers deterministically in different types of algorithms (like TOTP).

In this section, we will look at two standardized MAC algorithms that one can use nowadays: HMAC and KMAC.

3.5.1 HMAC, a hash-based message authentication code

The most widely used MAC is **HMAC** (for Hash-based MAC) invented in 1996 by M. Bellare, R. Canetti, and H. Krawczyk and specified in RFC 2104, FIPS PUB 198, and ANSI X9.71. HMAC, like its name indicates, is a way to use hash functions with a key. Using a hash function to build MACs is a popular concept as hash functions have widely available implementations, are fast in software, and also benefit from hardware support on most systems.

Remember that I have mentioned in chapter 2 that SHA-2 should not be used directly to hash secrets due to **length-extension attacks** (more on that at the end of this chapter). How does one figure out how to transform a hash function into a keyed function? This is what HMAC solves for us. Under the hood, HMAC follows these steps:

1. It first creates two keys from the main key: `k1 = k  ipad` and `k2 = k  opad` where `ipad` (inner padding) and `opad` (outer padding) are constants and is the symbol for the XOR operation.
2. It then concatenates a key `k1` with the message and hashes it.
3. The result is concatenated with a key `k2` and hashed one more time.
4. This produces the final authentication tag.

I illustrate this flow visually in figure 3.7.

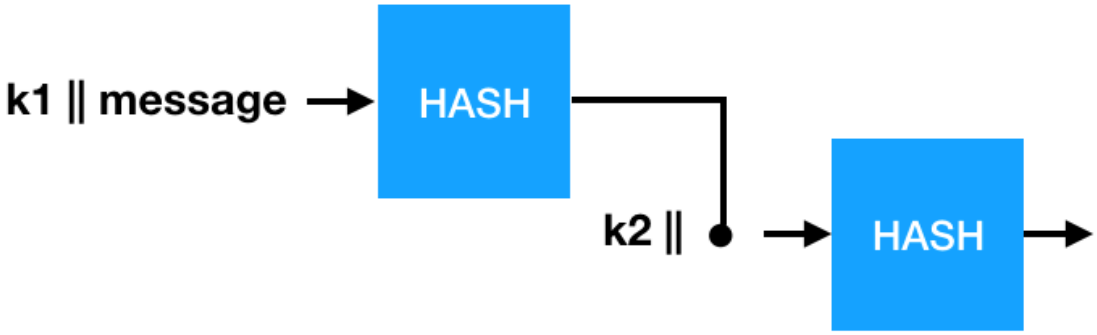


Figure 3.7 HMAC works by hashing the concatenation (`||`) of a key `k1` and the input message, and by then hashing the concatenation of a key `k2` with the output of the first operation. `k1` and `k2` are both deterministically derived from a secret key `k`.

Due to HMAC being customizable, the size of its authentication tag is dictated by the used hash function. For example, HMAC-SHA256 makes use of SHA-256 and produces an authentication tag of 256 bits, HMAC-SHA512 produces an authentication tag of 512 bits, and so on.

> **WARNING** **While one can truncate the output of HMAC to reduce its size, an authentication tag should be at minimum 128 bits as we've talked about earlier. This is not always respected and some applications will go as low as 64 bits due to explicitely handling a limited amount of queries. There are trade-offs with this approach and once again it is important to read the fine print before doing something non-standard.**

HMAC was constructed this way in order to facilitate proofs. In several papers, HMAC is proven to be secure against forgeries as long as the hash function underneath holds some good properties, which all cryptographically secure hash functions should hold. Due to this, HMAC can be used in combination with a large number of hash functions. Today, HMAC is mostly used with SHA-2.

3.5.2 KMAC, a hash based on cSHAKE

As SHA-3 is not vulnerable to length-extension attacks (this was actually a requirement for the SHA-3 competition), it makes little sense to use SHA-3 with HMAC instead of something like `SHA-3-256(key || message)` that would work well in practice.

This is exactly what **KMAC** is. It makes use of **cSHAKE**, the customizable version of the SHAKE extendable output function (XOF) you've seen in chapter 2. KMAC non-ambiguously encodes the MAC key, the input, and the output length requested (KMAC is some sort of extendable output MAC) and gives this to cSHAKE as an input to absorb. KMAC also passes a "KMAC" function name, and an user-defined customization string.

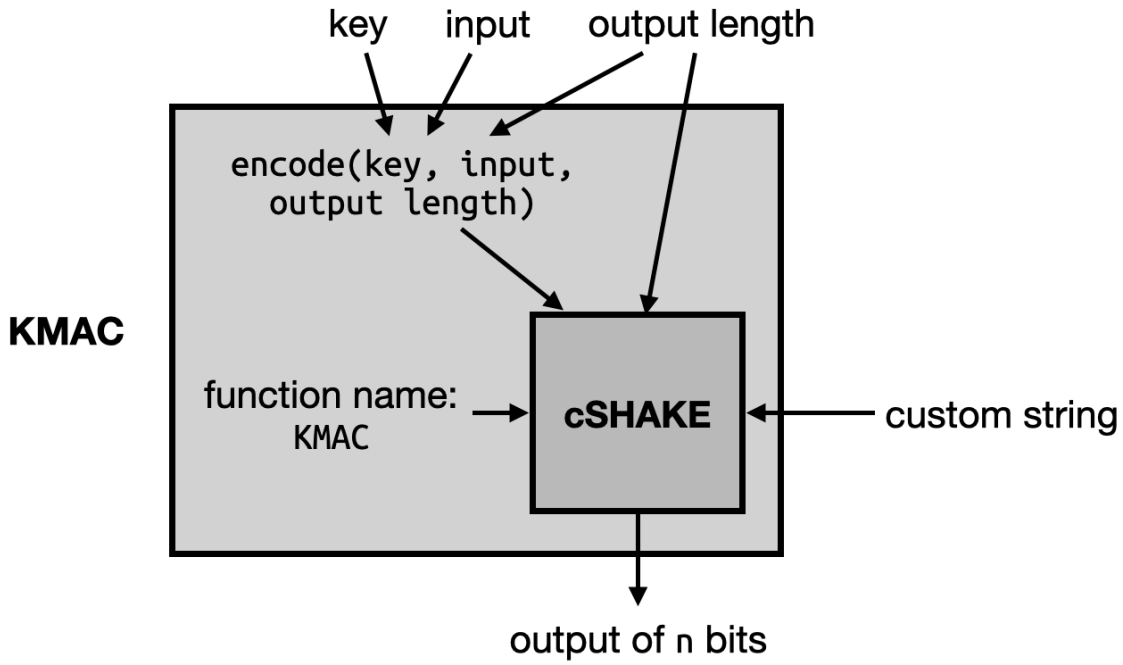


Figure 3.8 KMAC is simply a wrapper around cSHAKE. To use a key, it encodes (in a non-ambiguous way) the key, the input, and the output length as the input to cSHAKE.

Interestingly, since KMAC also absorbs the output-length requested, several calls with different output-length will provide totally different results (which is rarely the case for XOFs in general). This makes KMAC quite a versatile function in practice.

3.6 SHA-2 and length-extension attacks

We have mentioned several times that one shouldn't hash secrets with SHA-2 as it is not resistant to **length-extension attacks**. In this section, we aim to provide a simple explanation of what this attack is.

Let's go back to our introduction scenario, to the step where we attempted to simply use SHA-2 in order to protect the integrity of the cookie. Remember that it was not good enough, as the user can tamper with the cookie (for example, by adding an `admin=true` field) and recompute the hash over the cookie. Indeed, SHA-2 is a public function and nothing prevents the user from doing this. This is illustrated in figure 3.9.

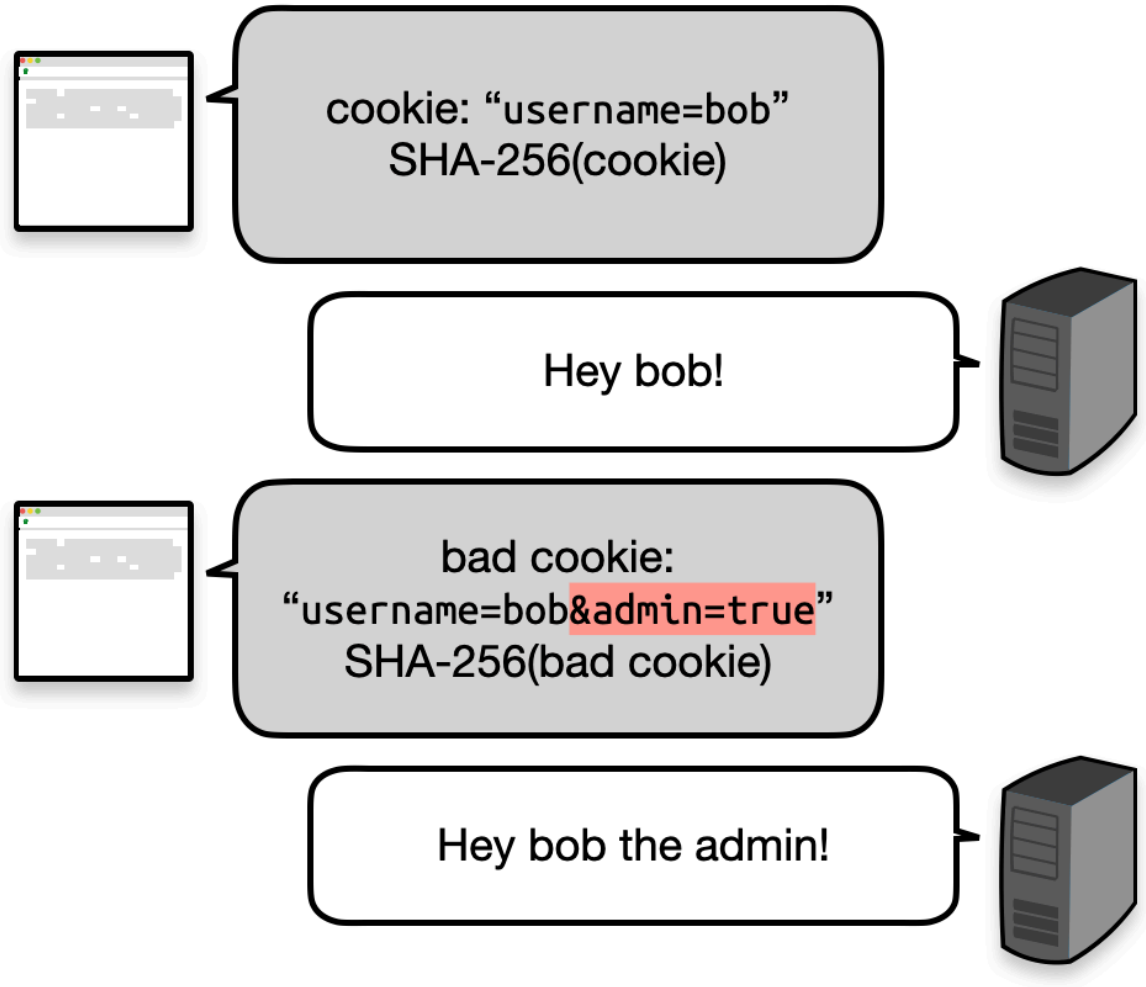


Figure 3.9 A webpage sends a cookie, followed by a hash of that cookie, to a user. The user is then required to send the cookie to authenticate him/herself in every subsequent request. Unfortunately, a malicious user can tamper with the cookie and recompute the hash, breaking the integrity check. The cookie is then accepted as valid by the webpage.

The next best idea could have been to add a secret key to what we hash. This way the user cannot recompute the digest as the secret key is required. Very much like a MAC. On receipt of the tampered cookie, the page will compute `SHA-256(key || tampered_cookie)` (where `||` represents the concatenation of the two values) and obtain something that won't match what the malicious user probably sent. We illustrate this in figure 3.10.

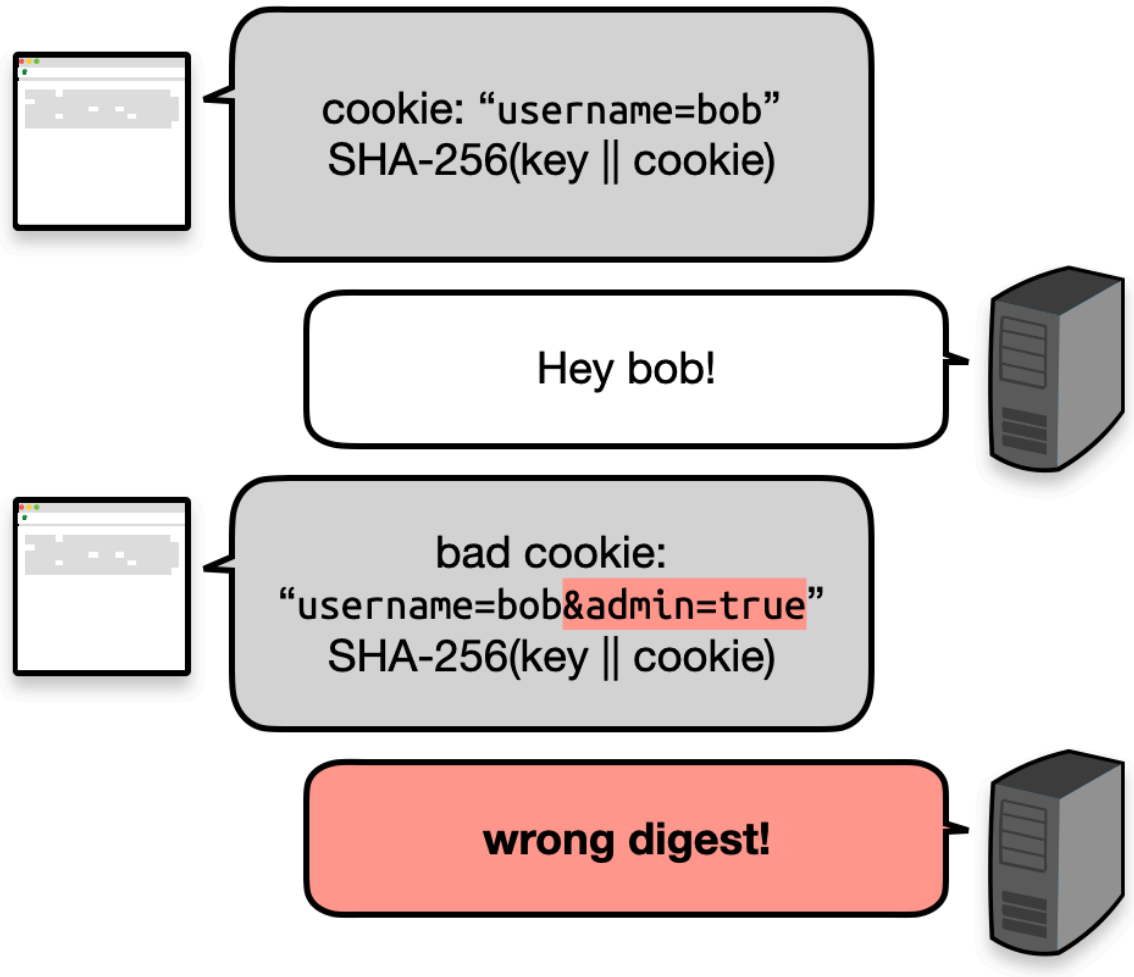


Figure 3.10 By using a key when computing the hash of the cookie, one could think that a malicious user (who wants to tamper with his cookie) wouldn't be able to compute the correct digest over the new cookie. We will see later that this is not true for SHA-256.

Unfortunately, SHA-2 has an annoying peculiarity: from a digest over an input, one can compute the digest of an input and more. What does this mean? Let's take a look at figure 3.11 where one uses SHA-256 as `SHA-256(secret || input1)`.

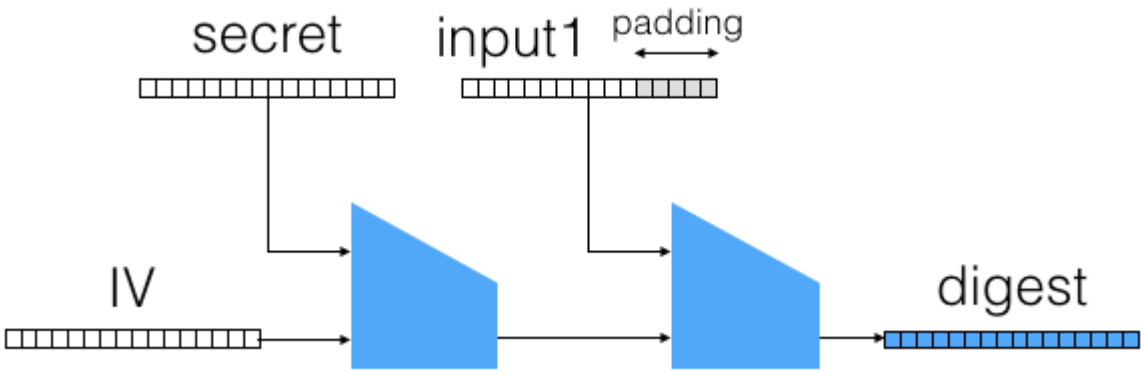


Figure 3.11 SHA-256 is used to hash a secret concatenated with a cookie (here named input1). Remember that SHA-256 works by having the Merkle–Damgård construction to iteratively call a compression function over blocks of the input, starting from an initialization vector (IV).

This figure is highly simplified, but imagine that `input1` is the string `user=bob`. Notice that the digest obtained is, effectively, the full intermediate state of the hash function at this point. Nothing prevents one from pretending that the `padding` section is part of the input and continuing the Merkle–Damgård dance. In diagram 3.12 we illustrate this attack, where one would take the digest and compute the hash of `input1 || padding || input2`. In our example, `input2` is `&admin=true`.

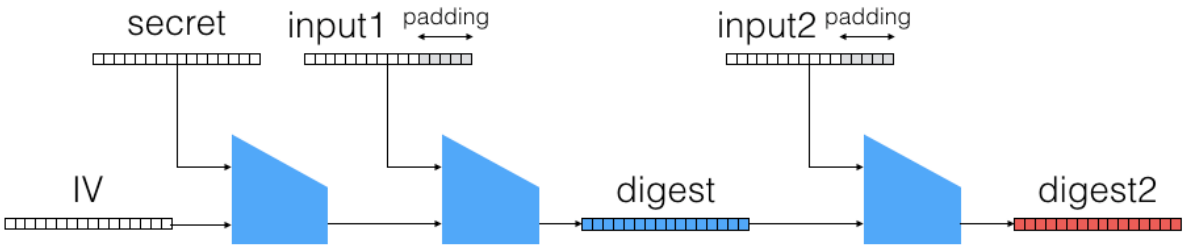


Figure 3.12 The output of the SHA-256 hash of a cookie (in blue) is used to extend the hash to more data, creating a hash (in red) of the secret concatenated with input1, the first padding bytes, and input2.

This vulnerability allows one to continue hashing, from a given digest, like the operation was not finished. Breaking our previous protocol as can be seen in figure 3.13.

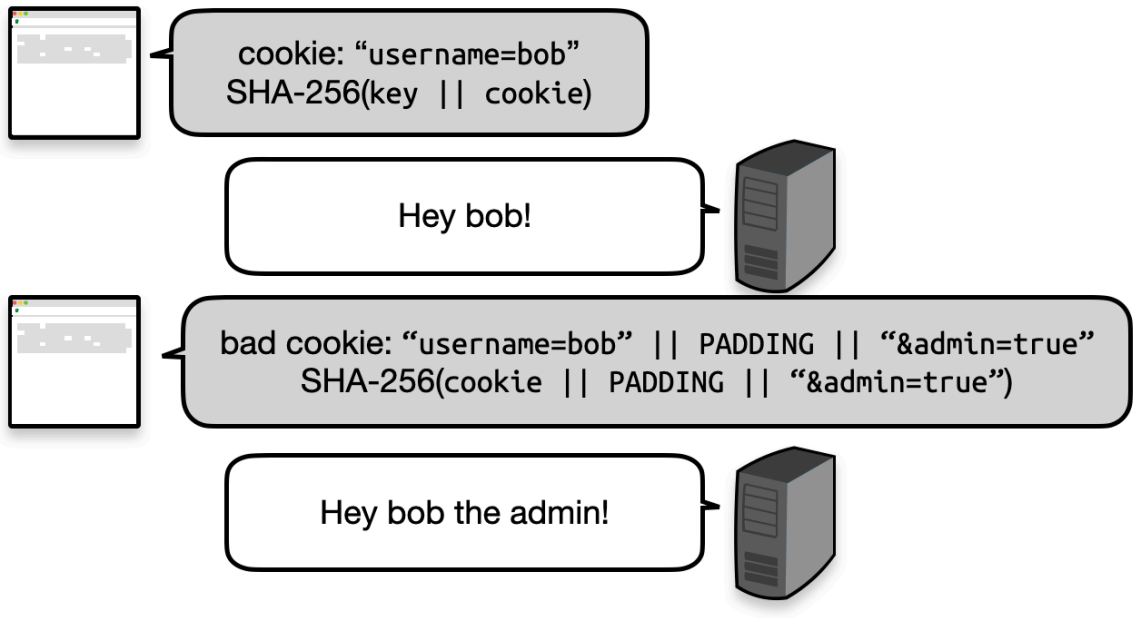


Figure 3.13 An attacker successfully uses a length-extension attack to tamper with his/her cookie and compute the correct hash using the previous hash.

The fact that the first padding now needs to be part of the input might prevent some protocols to be exploitable. Still, the smallest amount of change might reintroduce a vulnerability.

For this reason one should never, ever hash secrets with SHA-2. Of course there are several other ways to do it correctly (for example `SHA-256(k || message || k)` works) which is what HMAC provides. So use HMAC if you want to use SHA-2, use KMAC if you prefer SHA-3.

3.7 Summary

- Message authentication codes (MACs) are symmetric cryptographic algorithms that allow one or more parties who share the same key to verify the integrity and authenticity of messages.
  - To verify the authenticity of a message and its associated authentication tag, one can recompute the authentication tag of the message (and a secret key) and match the two authentication tags. If they differ, the message has been tampered with.
  - Always compare a received authentication tag with a computed one in constant-time.
- While MACs protect the integrity of messages, by default they do not detect when messages are being replayed.
- Standardized and well accepted MACs are the HMAC and the KMAC standards.
- One can use HMAC with different hash functions, in practice HMAC is often used with the SHA-2 hash function.
- Authentication tags should be of a minimum length of 128 bits to prevent collisions and forgery of authentication tags.
- Never use SHA-256 directly to build a MAC as it can be done incorrectly. Always use a function like HMAC to do this.

4 Authenticated encryption

This chapter covers

- Symmetric Encryption, a cryptographic primitive to hide communication from observers.
- Authenticated Encryption, the secure evolution of symmetric encryption.
- The popular authenticated encryption algorithms.
- Other types of symmetric encryption.

This is where it all began: the science of cryptography was first and foremost invented to fill our need to hide information. Encryption is what pre-occupied most of the early cryptographers: "How can we prevent observers from understanding our conversations?" While the science and its advances first bloomed behind closed door, benefiting the governments and their military only, it has now opened and spread throughout the world. Today, encryption is used everywhere to add a sense of privacy and security in the different aspects of our modern lives. In this chapter we'll find out what encryption really is, what types of problems it solves, and how today's applications makes use of this cryptographic primitive.

For this chapter you'll need to have read:

- Chapter 3 on message authentication codes.

4.1 What's a cipher?

It's like when you use slang to talk to your siblings about what you'll do after school so your mom doesn't know what you're up to.

– Natanael L. in 2020

Let's imagine that our two characters, Alice and Bob, want to exchange some messages

privately. In practice they have many mediums at their disposition: the mail, phones, the internet, and so on, and each of these mediums are by default insecure: the mailman could open their letters, the telecommunication operators can spy on their calls and text messages, internet service providers or any servers on the internet network that are in between Alice and Bob have access to the content of the packets being exchanged.

Without further ado, let's introduce Alice and Bob's savior: the **encryption algorithm**, also called "**cipher**." For now, let's picture this new algorithm as a black box that Alice can use to **encrypt** her messages to Bob. By "encrypting" a message, Alice transforms it into something that looks random.

This encryption algorithm takes:

- A **secret key**. It is crucial that this element is unpredictable (so random), and well protected, as the security of the encryption algorithm relies directly on the secrecy of the key. I will talk more about this in chapter 8 on secrets and randomness.
- A **plaintext**. This is what you want to encrypt. It can be some text, an image, a video, or anything that can be translated into bits.

This encryption process produces a **ciphertext**, which is the encrypted content. Alice can safely use one of the mediums listed previously to send that ciphertext to Bob. The ciphertext will look random to anyone who does not know the secret key, and no information about the content of the message (the plaintext) will be leaked. Once Bob receives this ciphertext, he can use a "decryption algorithm" to revert the ciphertext into the original plaintext.

Decryption takes:

- A **secret key**. This is the same secret key that Alice has used to create the ciphertext. Because the same key is used for both algorithms, we sometimes call the key a **symmetric key**. This is also why we also sometimes specify that we are using **symmetric encryption** and not just **encryption**.
- A **ciphertext**. This is the encrypted message Bob received from Alice.

The process then reveals the original **plaintext**. The full flow is illustrated in figure 4.1.

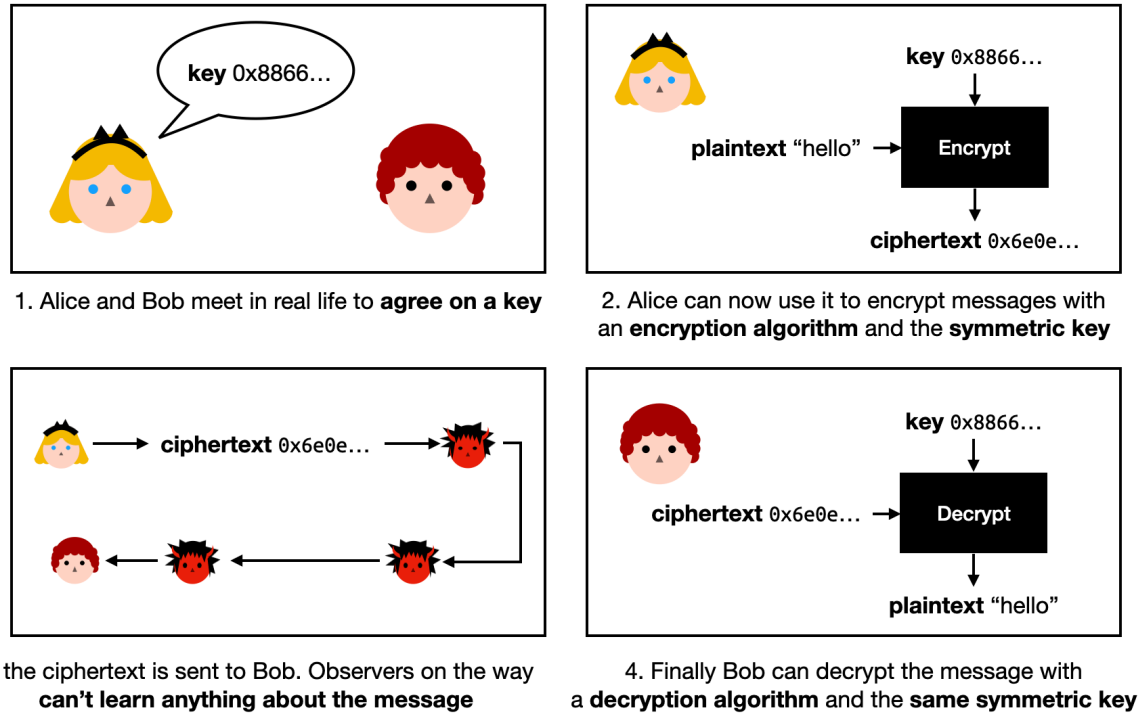


Figure 4.1 Alice (top right) can encrypt the plaintext "hello" with the key `0x8866…` (abbreviated hexadecimal). The ciphertext produced can be sent to Bob. Bob (bottom right) can decrypt the received ciphertext by using the same key and a decryption algorithm.

Encryption allows Alice to transform her message into something that looks random and that can safely be transmitted to Bob, and decryption allows Bob to revert the encrypted message back to the original message. This new cryptographic primitive provides **confidentiality** (or **secrecy**, or **privacy**) to their messages.

> **NOTE** How do Alice and Bob agree to use the same symmetric key? For now, we'll assume that one of them had access to an algorithm that generates unpredictable keys, and that they met in person to exchange the key. In practice, how to bootstrap such protocols with shared secrets is often one of the great challenges companies have to solve. In this book, you will see many different solutions to this problem.

Notice that I have yet to introduce what the title, authenticated encryption, refers to. I've only talked about encryption alone so far. While encryption alone is not secure (more about that later), I have to explain how it works before I can introduce the authenticated encryption primitive. So bare with me, as I first go over the main standard for encryption: the **Advanced Encryption Standard (AES)**.

4.2 The Advanced Encryption Standard (AES) block cipher

In 1997, the US National Institute of Standards and Technology (NIST) organized an open competition to replace the Data Encryption Standard (DES) algorithm, a standard for encryption at the time that was starting to show signs of age. The competition lasted 3 years, during which 15 different designs were submitted by teams of cryptographers from different countries. At the end of the competition, only one submission—Rijndael by Vincent Rijmen and Joan Daemen—was nominated as the winner. In 2001, the NIST released the Advanced Encryption Standard (AES) as part of the FIPS 197 publication. AES, the algorithm described in the AES standard, is still the main cipher used today.

In this section, I will explain how AES works.

4.2.1 How much security does AES provide?

AES offers three different versions: **AES-128** takes a key of 128 bits (16 bytes), **AES-192** takes a key of 192 bits (24 bytes), and **AES-256** takes a key of 256 bits (32 bytes). The length of the key, dictates the level of security: the bigger the stronger. Nonetheless, most applications make use of AES-128 as it provides enough security: **128 bits of security**.

The term **bit-security** is commonly used to indicate the security of cryptographic algorithms. For example with AES-128, it specifies that the best attack we know of would take around 2^{128} operations. This number is gigantic, and is the security level that most applications aim for.

> **NOTE** The fact that a 128-bit key provides 128 bits of security is specific to AES, it is not a golden rule: a 128-bit key used in some other algorithm could theoritcally provide less than 128-bit security. While a 128-bit key can provide less than 128-bit security, it will never provide more: there's always the brute-force attack. Trying all the possible keys would take at most 2^{128} operations, reducing the security to 128 bits at least.

How big is 2^{128}? Notice that the amount between two powers of 2 is doubled. For example 2^3 is twice as much as 2^2. So if 2^{100} operations are pretty much impossible to reach, imagine achieving double that (2^{101})! To reach 2^{128}, you have doubled your initial amount 128 times. In english, 2^{128} is *340 undecillion 282 decillion 366 nonillion 920 octillion 938 septillion 463 sextillion 463 quintillion 374 quadrillion 607 trillion 431 billion 768 million 211 thousand 456*. It is quite hard to imagine how big that number is, but you can assume that we will never be able to reach such a number in practice. We also didn't account for the amount of space required for any large and complex attacks to work, which is equally as enormous in practice.

It is foreseeable that AES-128 will remain secure for as long as we know, unless advances in cryptanalysis find a yet undiscovered vulnerability that would reduce the number of operations

needed to attack the algorithm.

4.2.2 The interface of AES

Looking at the interface of AES for encryption, we see the following:

- The algorithm takes a variable-length key, as discussed previously.
- It also takes a plaintext of exactly 128 bits.
- It outputs a ciphertext of exactly 128 bits.

Because AES encrypts a fixed-size plaintext, we call it a **block cipher**. Some other ciphers can encrypt arbitrary-length plaintexts, as you will see later in this chapter.

The decryption operation is exactly the reverse of this: it takes the same key, a ciphertext of 128 bits and returns the original 128-bit plaintext. Effectively, decryption reverts the encryption. This is possible, because the encryption and decryption operations are **deterministic**: they will produce the same results no matter how many times you call them.

In technical terms, a block cipher with a key is a **permutation**: it maps all the possible plaintexts to all possible ciphertexts (see example in figure 2.13). Changing the key changes that mapping. A permutation is also reversible: from a ciphertext, you have a map back to its corresponding plaintext (otherwise decryption wouldn't be very useful).

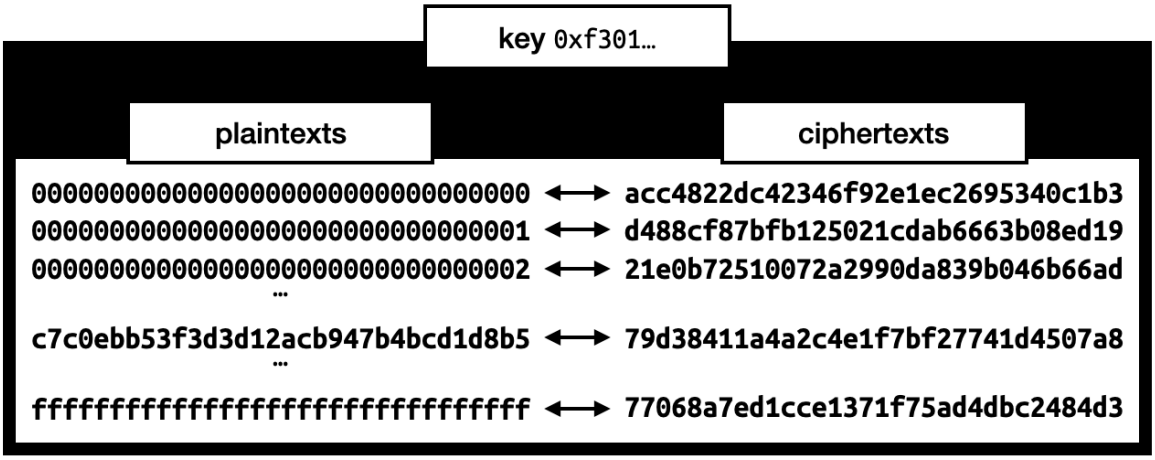


Figure 4.2 A cipher with a key can be seen as a permutation: it maps all the possible plaintexts to all the possible ciphertexts.

Of course, we do not have the room to list all the possible plaintexts and their associated ciphertexts. That would be 2^{128} mappings for a 128-bit block cipher. Instead, we design constructions like AES which behave like permutations, and are randomized by a key. We say that they are **pseudo-random permutations (PRPs)**.

4.2.3 The internals of AES

Let's dig a bit deeper into the guts of AES, to see what's inside. Note that AES sees the **state** of the plaintext, during the encryption process, as a 4-by-4 matrix of bytes (as can be seen in figure 4.3).

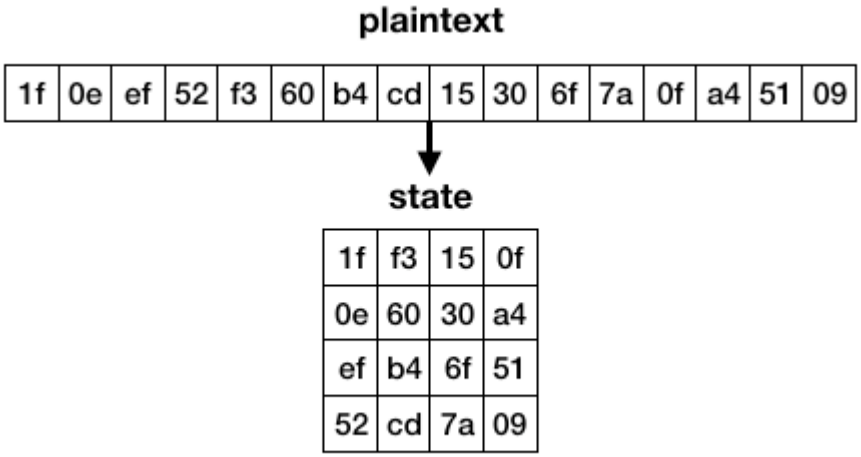


Figure 4.3 When entering the AES algorithm, a plaintext of 16 bytes gets transformed into a 4-by-4 matrix. This state is then encrypted, and finally transformed into a 16-byte ciphertext.

This doesn't really matter in practice, but this is how AES is defined. Under the hood, AES works like many similar symmetric cryptographic primitives called **block ciphers** (as they are ciphers that encrypt fixed-sized blocks).

AES has a round function that it iterates several times starting on the original input (the plaintext). I illustrate this overview in figure 4.4.

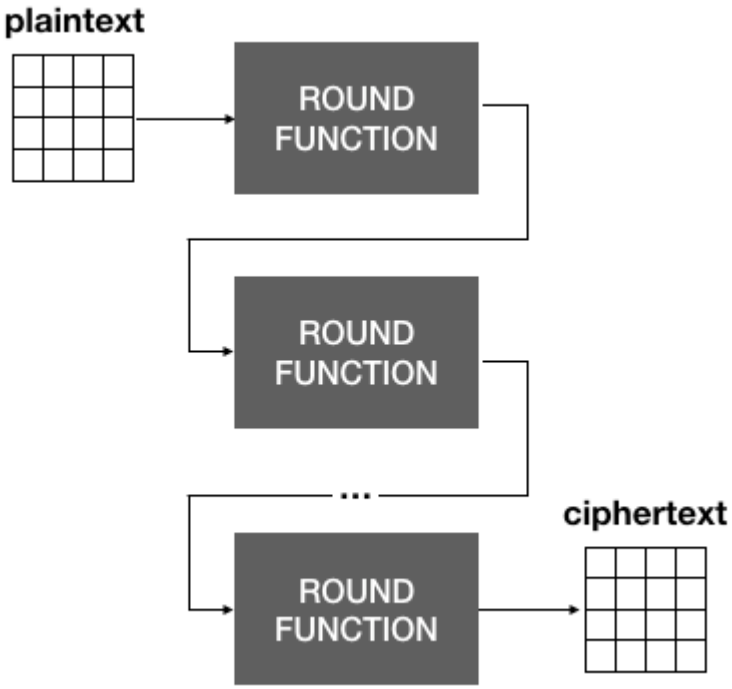


Figure 4.4 AES iterates a round function over a state in order to encrypt it. The round function takes several arguments, including a secret key. This is missing from the diagram for simplicity.

Each call to the round function transforms the state further, eventually producing the ciphertext. Each round uses a different round key, which are derived from the main symmetric key (during what is called a key schedule). This allows the slightest change in the bits of the symmetric key to give a completely different encryption (a principle called **diffusion**).

The round function is comprised of multiple operations that mix and transform the bytes of the state. The round function of AES specifically makes use of 4 different subfunctions. While we will shy away from explaining exactly how they work (you can find this information in any book about AES), they are named `SubBytes`, `ShiftRows`, `MixColumns` and `AddRoundKey`. The first three are easily reversible (you can find the input from the output of the operation) but the last one is not, as it performs an "exclusive OR" (XOR) with the round key and the state (and thus needs the knowledge of the round key to be reversed). I illustrate what goes into a round in figure 4.5.

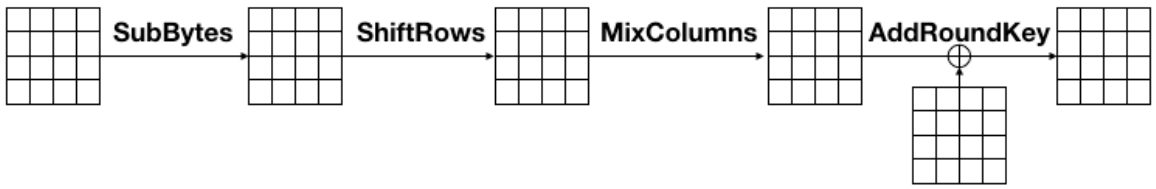


Figure 4.5 A typical round of AES. (The first and last rounds omit some operations.) 4 different functions transform the state. They are all reversible as it is required for decryption. The addition sign inside a circle is the symbol for the XOR operation.

The number of iterations of the round function in AES was chosen to thwart cryptanalysis, which are usually practical on a reduced number of rounds. For example, extremely efficient total breaks (attacks that recover the key) exist on 3-round variants of AES-128. By iterating many times, the cipher transforms a plaintext into something that looks nothing like the original plaintext. The slightest change in the plaintext also gives out a completely different ciphertext. This principle is called the **avalanche effect**.

> **NOTE** Real-world cryptographic algorithms are typically compared by the security, size and speed they provide. We've already talked about the security and size of AES: its security depends on the key size, and it can encrypt 128-bit blocks of data at a time. Speed-wise, AES has been implemented in hardware by many CPU vendors: AES-New Instructions (AES-NI) is a set of instructions implemented by Intel and AMD CPUs that can be used to encrypt and decrypt with AES. These special instructions make AES extremely fast in practice.

One question that you might still have is "how do I encrypt more or less than 128 bits with AES?" I'll answer this next.

4.3 The encrypted penguin and the CBC mode of operation

Now that we have introduced the AES block cipher, and explained a bit about its internals, let's see how to use it in practice. The problem with a block cipher, is that it can only encrypt a block by itself. To encrypt something that is not exactly 128 bits, a **padding** as well as a **mode of operation** must be used. So let's see what these two words are about.

Imagine that you want to encrypt a long message. Naively, you could divide the message into blocks of 16-byte (the block size of AES). Then, if the last block of plaintext is smaller than 16 bytes, you could append some more bytes at the end until the plaintext becomes 16-byte long. This is what padding is about!

There are several ways to specify how to choose these "padding bytes", but the most important aspect of padding is that it must be reversible: once a ciphertext is decrypted, one should be able to remove the padding to retrieve the original unpadded message. For example, simply adding random bytes wouldn't work, as you wouldn't be able to discern if the random bytes were part of the original message or not.

The most popular padding mechanism is often referred to as "PKCS#7 padding," due to first appearing in the PKCS#7 standard published by the RSA company in the end of the '90s. The PKCS#7 padding specifies one rule: the value of each padding byte must be set to the length of the required padding. What if the plaintext is already 16 bytes? Then add a full block of padding set to the value 16. I illustrate this visually in figure 2.9.

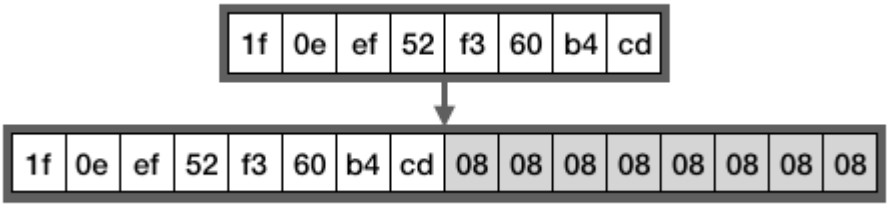


Figure 4.6 If a plaintext is not a multiple of the block size, it is padded with the length needed to reach a multiple of the block size. In the example, the plaintext is 8 bytes, so 8 more bytes (containing the value 8) are used to pad the plaintext up to the 16 bytes required for AES.

To remove the padding, you can easily check the value of the last byte of plaintext and interpret it as the length of padding to remove.

Now, there's one big problem I need to talk about. So far, to encrypt a long message, you've just divided it into blocks of 16 bytes (and perhaps you padded the last block). This naive way is called the **electronic codebook (ECB)** mode of operation. As you've learned, encryption is deterministic, and so encrypting the same block of plaintext twice will lead the same ciphertext. This means that by encrypting each block individually, the resulting ciphertext might have repeating patterns.

This might seem fine, but allowing these repeatitions lead to many problems, the most obvious one is that they leak information about the plaintext. The most famous illustration of this is the ECB penguin, pictured in figure 4.7.

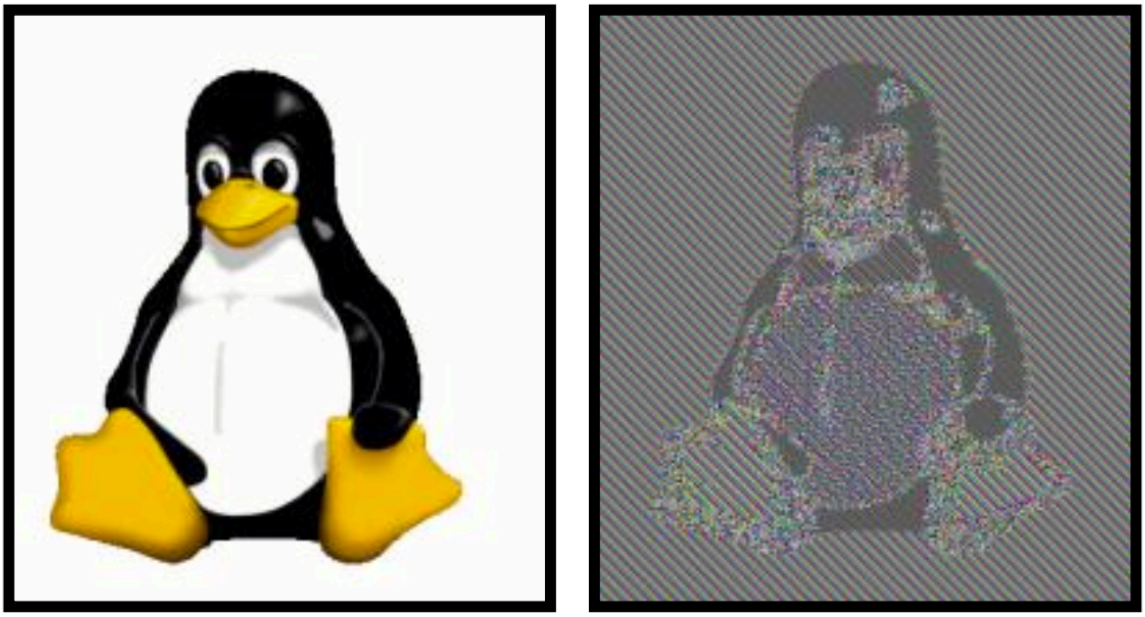


Figure 4.7 The famous ECB penguin is an encryption of an image of a penguin using the ECB mode of operation. As ECB does not hide repeating patterns, one can guess just by looking at the ciphertext what was originally encrypted. (Image taken from Wikipedia.)

To do this, better modes of operation exist that "randomize" the encryption. One of the most popular modes of operation for AES is the **Cipher Block Chaining** (CBC). CBC works for any deterministic block cipher (not just AES) by taking an additional value called an initialization vector (IV) to randomize the encryption. Because of this, the IV is the length of the block size (16 bytes for AES) and must be random and unpredictable.

To encrypt with the CBC mode of operation, start by generating a random IV of 16-byte (we will see in chapter 8 how to do this), then XOR the generated IV with the first 16-byte of plaintext before encrypting them. This effectively randomize the encryption. Indeed, if the same plaintexts is encrypted twice but with different IVs, the mode of operation will render two different ciphertexts.

If there is more plaintext to encrypt, use the previous ciphertext (like we used the IV previously) to XOR it with the next block of plaintext before encrypting it. This effectively randomizes the next block of encryption as well. Remember, the encryption of something is unpredictable and should be as good as the randomness we used to create our real IV. I illustrate CBC encryption in figure 4.8.

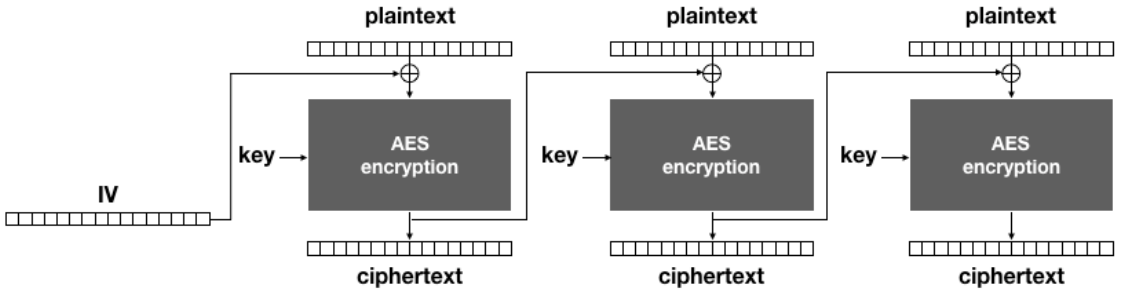


Figure 4.8 The CBC mode of operation with AES. To encrypt, a random initialization vector (IV) is used in addition to a padded plaintext (split in multiple blocks of 16 bytes).

To decrypt with the CBC mode of operation, reverse the operations. As the IV is needed, it must be transmitted in clear text along with the ciphertext. Since the IV is supposed to be random, no information is leaked by observing the value. I illustrate CBC decryption in figure 4.9.

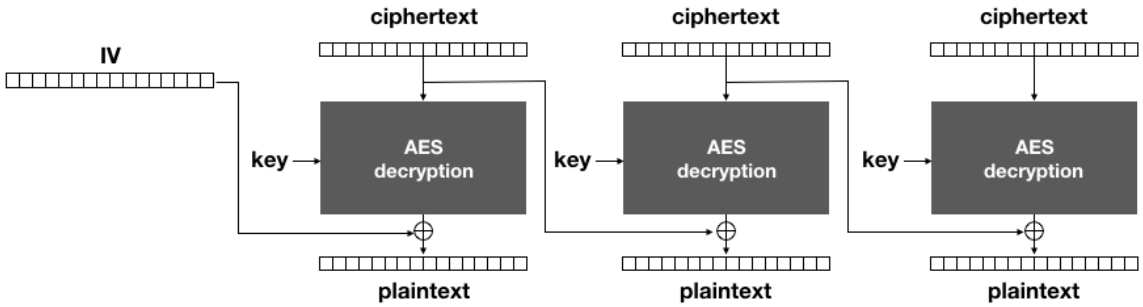


Figure 4.9 The CBC mode of operation with AES. To decrypt, the associated initialization vector (IV) is required.

Additional parameters, like IVs, are prevalent in cryptography. Yet, they are often poorly understood and a great source of vulnerabilities. With the CBC mode of operation, an IV needs to be **unique** (it cannot repeat), as well as **unpredictable** (it really needs to be random). These requirements can fail for a number of reasons. Since developers are often confused with IVs, some cryptographic libraries have removed the possibility to specify an IV when encrypting with CBC, and automatically generate one randomly.

> **NOTE** When an IV repeats or is predictable, the encryption becomes deterministic again and a number of clever attacks become possible. This was the case with the famous BEAST attack on the TLS protocol. Note also that other algorithms might have different requirements for IVs. This is why it is always important to read the manual. Dangerous details lie in the fine print.

Note that a mode of operation and a padding are still not enough to make a cipher usable, as you're about to see in the next section.

4.4 A lack of authenticity, hence AES-CBC-HMAC

We have failed to address one fundamental flaw so far: the ciphertext as well as the IV (in the case of CBC) can still be modified by an attacker. Indeed, there's no integrity mechanism to prevent that. Changes in the ciphertext or IV might have unexpected changes in the decryption.

For example, in AES-CBC (AES used with the CBC mode of operation), an attacker can flip specific bits of a plaintext by flipping bits in its IV and ciphertext. I illustrate this attack in figure 4.10.

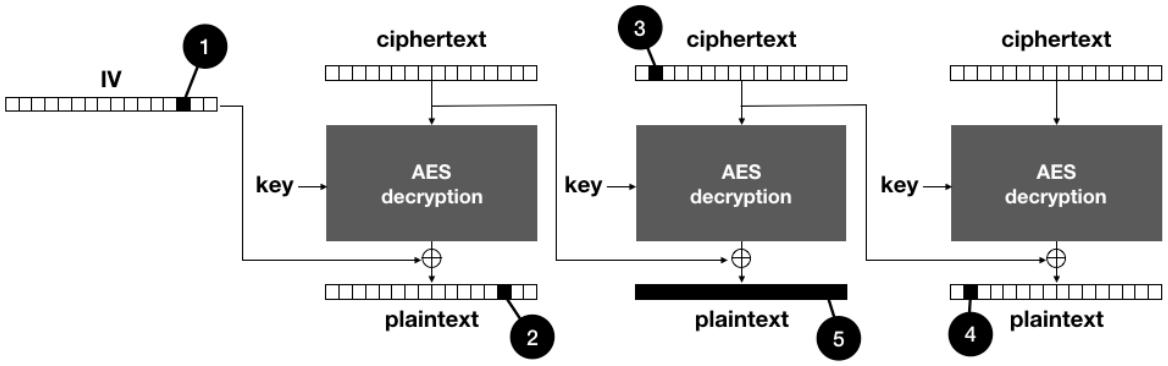


Figure 4.10 An attacker that can intercept an AES-CBC ciphertext can do the following: (1) Since the IV is public, flipping a bit (from 1 to 0, for example) of the IV also (2) flips a bit of the first block of plaintext. (3) Modifications of bits can happen on the ciphertext blocks as well (4) which are reflected on the next block of decrypted plaintext. (5) Note that tampering with the ciphertext blocks has the direct effect of scrambling the decryption of that block.

Consequently, a cipher or a mode of operation must not be used as-is. They lack integrity protection. This property ensures that a ciphertext and its associated parameters (here the IV) cannot be modified without triggering some alarms.

To prevent modifications on the ciphertext, the **message authentication codes (MACs)** that we've seen in chapter 3 can be used! For AES-CBC, **HMAC** (with the **SHA-256** hash function) is usually used in combination to provide integrity. The MAC is applied after padding the plaintext and encrypting it, and over both the ciphertext and the IV, otherwise an attacker can still modify the IV without being caught.

> **NOTE** This construction is called Encrypt-then-MAC. The alternatives (like MAC-then-Encrypt) can sometimes lead to clever attacks (like the famous Vaudenay padding oracle attack) and are thus avoided in practice.

The authentication tag created can be transmitted along with the IV and the ciphertext. Usually, all are concatenated together as illustrated in figure 4.11. In addition, it is best practice to use different keys for AES-CBC and HMAC.

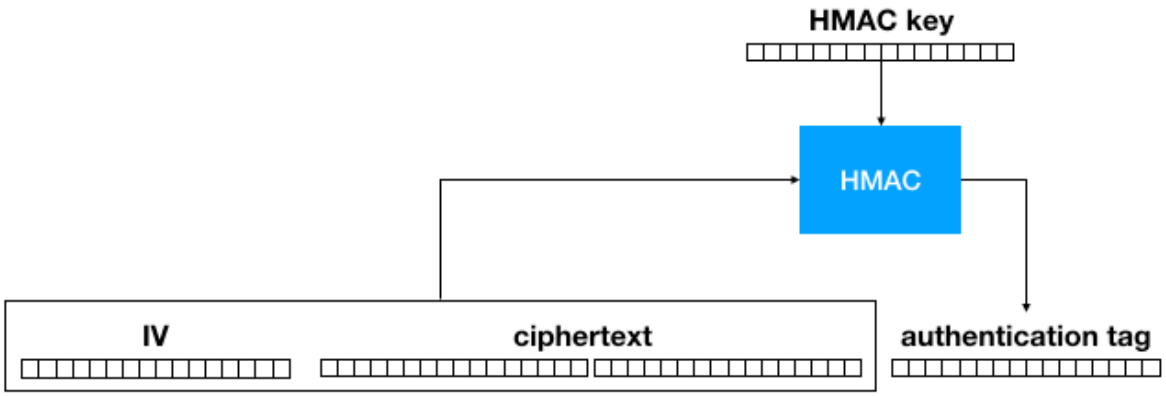


Figure 4.11 The AES-CBC-HMAC construction produces three arguments that are usually concatenated in the following order: the public IV, the ciphertext, the authentication tag.

Prior to decryption, the tag needs to be verified (in constant-time, as you've seen in chapter 3). The combination of all of these algorithms is referred to as **AES-CBC-HMAC** and has been one of the most widely used authenticated encryption modes, that is until more recent all-in-one constructions started being adopted.

> **WARNING** AES-CBC-HMAC is not the most developer-friendly construction, it is often poorly implemented and has some dangerous pitfalls when not used correctly (for example the IV of each encryption must be unpredictable). I have spent a few pages introducing this algorithm as it is still widely used, and still works, but I recommend against using it in favor of the more recent constructions I will introduce next.

4.5 All-in-one constructions: authenticated encryption

The history of encryption is not pretty. Not only has it been poorly understood that encryption without authentication is dangerous, but mis-applying authentication has also been a systemic mistake made by developers. For this reason, a lot of research has emerged seeking to standardize all-in-one constructions that simplify the use of encryption for developers. In the rest of this section I introduce this new concept as well as two widely adopted standards: AES-GCM and ChaCha20-Poly1305.

4.5.1 What's authenticated encryption with associated data (AEAD)?

The most current way of encrypting data is to use an all-in-one construction called **Authenticated Encryption with Associated Data** (AEAD). The construction is extremely close to what AES-CBC-HMAC provides as it also offers confidentiality of your plaintexts while detecting any modifications that could have occured on the ciphertexts. What's more, it provides a way to authenticate **associated data** (AD).

This **associated data** argument is optional and can be empty, or it can also contain metadata that is relevant to the encryption and decryption of the plaintext. This data will not be encrypted, and is either implied or transmitted along with the ciphertext.

In addition, the ciphertext's size is larger than the plaintext, as it now contains an additional authentication tag (usually appended to the end of the ciphertext).

To decrypt the ciphertext, we are required to use the same implied or transmitted **associated data**. The result is either an error, indicating that the ciphertext was modified in transit, or the original plaintext. I illustrate this new primitive in figure 4.12.

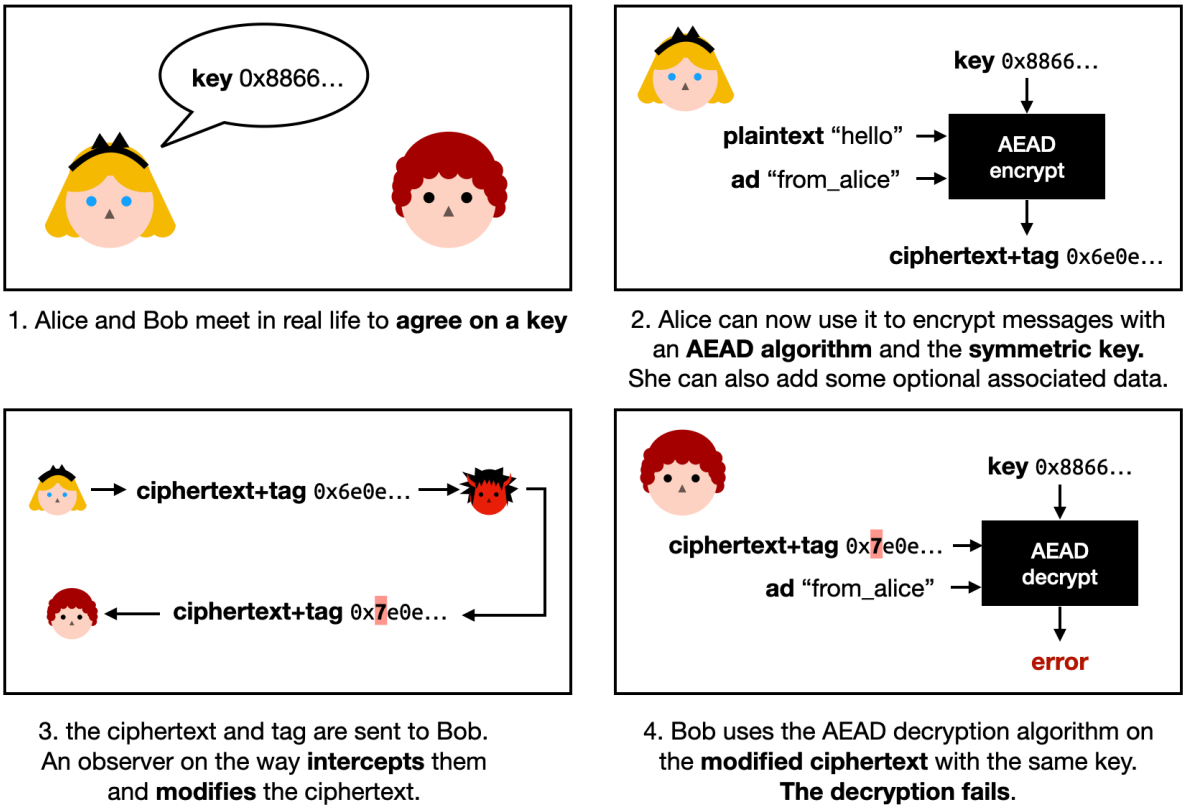


Figure 4.12 Both Alice and Bob meet in-person to agree on a shared key. Alice can then use an AEAD encryption algorithm with the key to encrypt her messages to Bob. She can optionally authenticate some associated data (`ad`), for example, the sender of the message. After receiving the ciphertext and the authentication tag, Bob can decrypt it using the same key and associated data. If the associated data is incorrect, or the ciphertext was modified in transit, the decryption will fail.

Let's see how to use a cryptographic library to encrypt and decrypt with an authenticated encryption primitive. For this, we'll use the Javascript programming language and the **Web Crypto API**, an official interface supported by most browsers that provides low-level cryptographic functions.

Listing 4.1 encrypt.js

```
let config = {
    name: 'AES-GCM',
    length: 128    ❶
};
let keyUsages = ['encrypt', 'decrypt'];
let key = await crypto.subtle.generateKey(config,
  false, keyUsages);

let iv = await crypto.getRandomValues(new Uint8Array(12));   ❷

let te = new TextEncoder();
let ad = te.encode("some associated data");   ❸
let plaintext = te.encode("hello world");

let param = {
    name: 'AES-GCM',
    iv: iv,
    additionalData: ad
};
let ciphertext = await crypto.subtle.encrypt(param,
  key, plaintext);

let result = await window.crypto.subtle.decrypt(param,
  key, ciphertext);   ❹
new TextDecoder("utf-8").decode(result);
```

❶ Generate a 128-bit key for 128 bits of security.

❷ We generate a 12-byte IV randomly.

❸ We uses some associated data to encrypt our plaintext. Decryption must use the same IV and associated data.

❹ Decryption will throw an exception if the IV, ciphertext, or associated data were tampered with.

Note that Web Crypto API is a low-level API, and as such does not help the developer to avoid mistakes. For example, it lets us specify an IV, which is a dangerous pattern.

In this example, I used AES-GCM which is the most widely used AEAD. Next, let's talk more about this AES-GCM!

4.5.2 The AES-GCM AEAD

The most widely used AEAD is AES with the Galois/Counter Mode also abbreviated **AES-GCM**. It was designed for high performance, by taking advantage of hardware support for AES, and by using a MAC (GMAC) that can be implemented efficiently.

AES-GCM is included in NIST's Special Publication 800-38D since 2007, is FIPS approved, and was also part of NSA's suite B (a suite of algorithms that can be used by the US government to protect unclassified and classified information). It is the main cipher used in many protocols, including several versions of the TLS protocol which is used to secure connections to websites on the internet. Effectively, we can say that AES-GCM encrypts the web.

AES-GCM combines the Counter Mode (CTR) mode of operation, with the GMAC message authentication code. Let's see how CTR with AES works first:

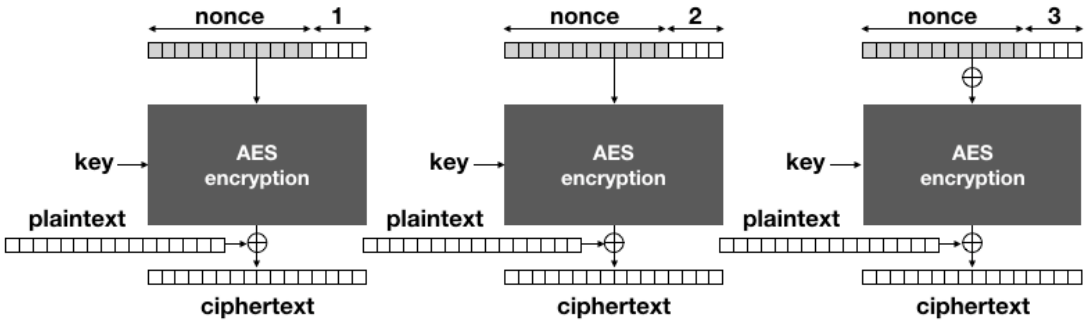


Figure 4.13 AES-CTR. A unique nonce is concatenated with a counter and encrypted to produce a "keystream." The keystream is then XORed with the actual bytes of the plaintext.

AES-CTR uses AES to encrypt a **nonce** concatenated with a number (starting at 1) instead of the plaintext. This additional argument, a **nonce** for "number once," serves the same purpose as an IV: it allows the mode of operation to randomize the AES encryption. The requirements are a bit different from the IV of CBC mode: a nonce needs to be **unique**, but not unpredictable. Once this 16-byte block is encrypted, the result is called a **keystream** and it is XORed with the actual plaintext to produce the encryption.

> **NOTE** Like IVs, nonces are a common term in cryptography, and they are found in different cryptographic primitives. They can have different requirements, although the name often indicates that it should not repeat. But as usual, what matters is what the manual said, not what the name of the argument implies. Indeed, the nonce of AES-GCM is sometimes referred to as an IV.

In the diagram, the nonce is 96-bit (12-byte) and takes most of the 16-byte to be encrypted. The 32 bits (4 bytes) left serves as a counter, starting from 1 and incremented for each block encryption until it reaches its maximum value at $2^{4\times8}-1 = 4{,}294{,}967{,}295$. This means that at most 4,294,967,295 blocks of 128 bits can be encrypted with the same nonce (so less than 69 gigabytes).

If the same nonce is used twice, the same keystream is created. By XORing the two ciphertexts together, the keystream is canceled and one can recover the XOR of the two plaintext. This can be devastating, especially if you have some information about the content of one of the two plaintexts.

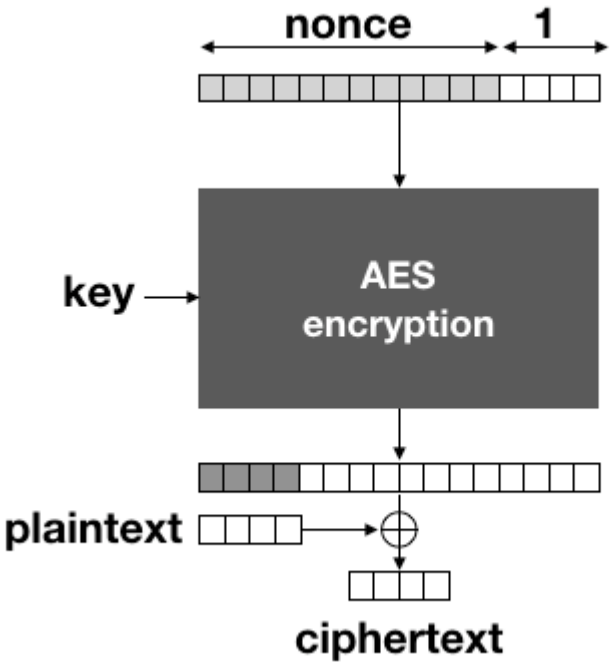


Figure 4.14 If the keystream of AES-CTR is longer than the plaintext, it is truncated to the length of the plaintext prior to XORing it with the plaintext. This permits AES-CTR to work without a padding.

An interesting aspect of Counter Mode is that no padding is required. We say that it turns a block cipher (AES) into a **stream cipher**: it encrypts the plaintext bytes by bytes.

NOTE Stream ciphers are another category of ciphers. They are different than block ciphers because they can be used directly to encrypt a ciphertext by XORing it with a keystream. No need for a padding or a mode of operation, allowing the ciphertext to be of the same length as the plaintext. In practice, there isn't much difference between these two categories of ciphers since block ciphers can easily be transformed into stream ciphers via the CTR mode of operation. But in theory, block ciphers have the advantage that they can be useful to construct other categories of primitives, similar to what you've seen in chapter 2 with hash functions. This is also a good moment to note that by default encryption doesn't (or badly) hides the length of what you are encrypting. Because of this, the use of compression before encryption can lead to attacks if an attacker can influence parts of what is being encrypted.

The second part of AES-GCM is **GMAC**. It is a MAC constructed from a keyed-hash **GHASH**. In technical terms, GHASH is an almost-XOR universal hash (AXU) also called a difference unpredictable function (DUF). The requirement of such a function is weaker than a hash. For example, an AXU does not need to be collision resistant. Thanks to this, GHASH can be significantly faster.

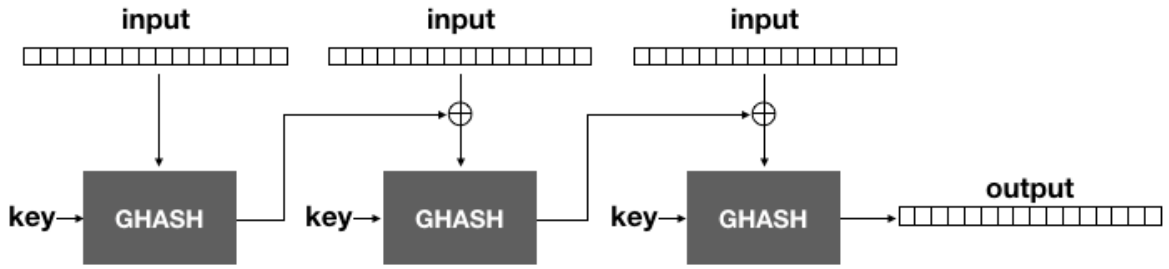


Figure 4.15 GHASH takes a key, and absorbs the input block by block in a manner resembling CBC mode. It produces a digest of 16 bytes.

To hash something with GHASH, the input is broken in blocks of 16-bytes and hashed in a similar way as CBC mode. As this hash takes a key as input, it can theoretically be used as a MAC, but only once (otherwise the algorithm breaks)—It's a one-time MAC.

As this is not ideal for us, we use a technique due to Wegman-Carter to transform GHASH into a many-time MAC.

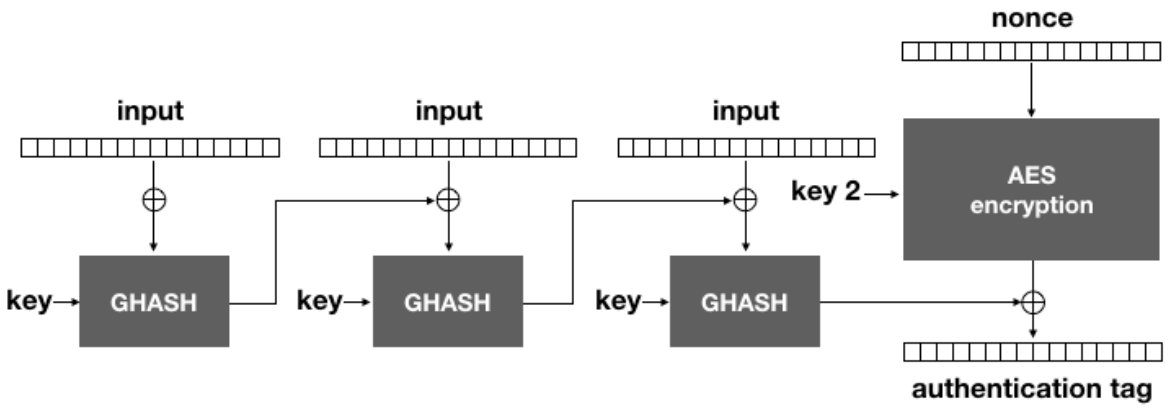


Figure 4.16 GMAC uses GHASH with a key to hash the input, then encrypts it with a different key and AES-CTR to produce an authentication tag.

GMAC is effectively the encryption of the GHASH output with AES-CTR (and a different key). The nonce must be **unique**, otherwise clever attacks can recover the authentication key used by GHASH (which would be catastrophic and would allow easy forgery of authentication tags).

Finally, **AES-GCM** can be seen as an intertwined combination of CTR mode and GMAC, similar to the Encrypt-then-MAC construction we've previously discussed.

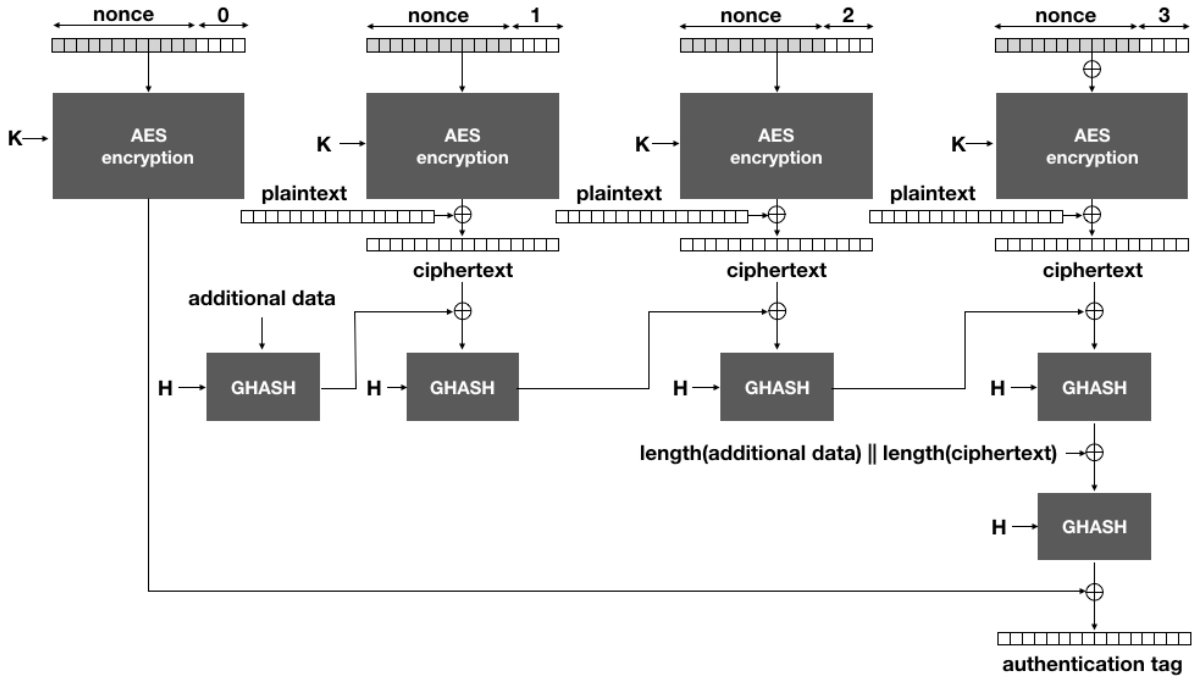


Figure 4.17 AES-GCM works by using AES-CTR with a symmetric key K to encrypt the plaintext, and using GMAC to authenticate the associated data and the ciphertext using an authentication key H.

The counter starts at 1 for encryption (unlike AES-CTR that starts its counter at 0), leaving the 0 counter for encrypting the authentication tag created by GHASH. GHASH in turns take an independent key H which is the encryption of the all-zero block with a key K. This way one does not need to carry two different keys, as the key K suffices to derive the other one.

As I've said previously, the 12-byte nonce of AES-GCM needs to be unique, and thus to never repeat. Notice that it doesn't need to be random, consequently some people like to use it as a **counter**, starting it at 1 and incrementing it for each encryption. In this case, one must use a cryptographic library that lets the user to choose the nonce. This allows one to encrypt $2^{12\times8}$-1 messages before reaching the maximum value of the nonce. Suffice to say, this is an impossible number of messages to reach in practice.

On the other hand, having a counter means that one needs to keep state. If a machine crashes at the wrong time, it is possible that nonce-reuse could happen. For this reason, it is sometimes preferred to have a **random nonce**. Actually, some libraries will not let developers choose the nonce and will generate them at random. Doing this should avoid repetition with probabilities so high that they shouldn't happen in practice. Yet, the more messages are encrypted, the more nonces are used, and the higher the chances of getting a collision become. Because of the **birthday bound** we talked about in chapter 2, it is recommended not to encrypt more than $2^{92/3} \approx 2^{30}$ messages with the same key (when generating nonces randomly).

NOTE 2^{30} messages is quite a large number of messages. It might never be reached in many scenarios, but real-world cryptography often pushes the limit of what is considered reasonable. Some long-lived systems need to encrypt many many messages per second, eventually reaching these limits. Visa, for example, process 150 million transactions per day, if it needs to encrypt them with a unique key it would reach the limit of 2^{30} messages in only a week. In these extreme cases, rekeying (changing the key used to encrypt) can be a solution, other research exist to improve the maximum number of messages that can be encrypted with the same key.

4.5.3 ChaCha20-Poly1305

The second AEAD I will talk about is **ChaCha20-Poly1305**. It is the combination of two algorithms, the ChaCha20 stream cipher and the Poly1305 MAC, both designed separately by Daniel J. Bernstein. Both algorithms were designed to be fast in software, contrary to AES which is slow when hardware support in unavailable. In 2013 Google standardized the ChaCha20-Poly1305 AEAD, in order to make use of it in Android mobile phones relying on low-end processors. Nowadays it has been widely adopted by internet protocols like OpenSSH, TLS and Noise.

ChaCha20 is a modification of the **Salsa20** stream cipher, which was originally designed by Daniel J. Bernstein around 2005. It was one of the nominated algorithms in the ESTREAM competition.

Like all stream cipher, the algorithm produces a **keystream**—a series of random bytes of the length of the plaintext. It is then XORed with the plaintext to create the ciphertext. To decrypt, the same algorithm is used to produce the same keystream, which is XORed with the ciphertext to give back the plaintext. I illustrate both flows in figure 4.18.

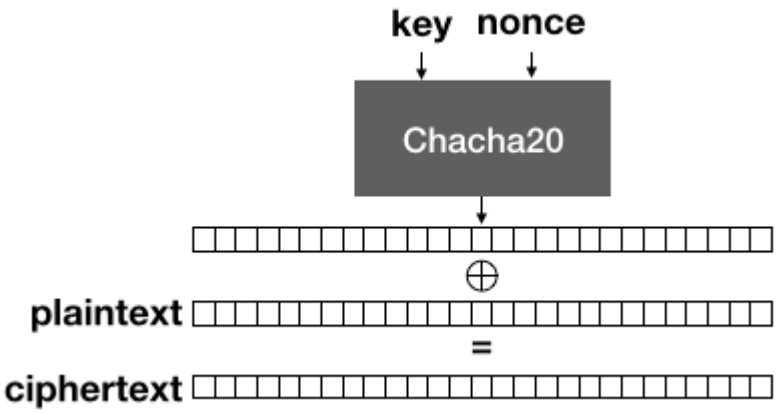


Figure 4.18 ChaCha20 works by taking a symmetric key and a unique nonce. It then generates a keystream that is XORed with the plaintext (or ciphertext) to produce the ciphertext (or plaintext). The encryption is length-preserving, as the ciphertext and the plaintext are of the same length.

Under the hood, ChaCha20 generates a keystream by repeatedly calling a **block function** to produce many 64-byte blocks of keystream. The block function takes:

- a 256-bit (32-byte) key, like AES-256
- a 92-bit (12-byte) nonce, like AES-GCM
- a 32-bit (4-byte) counter, like AES-GCM

The process to encrypt is the same as with AES-CTR:

1. Run the block function, incrementing the counter every time, until enough keystream is produced.
2. Truncate the keystream to the length of the plaintext.
3. XOR the keystream with the plaintext.

I illustrate this flow in figure 4.19.

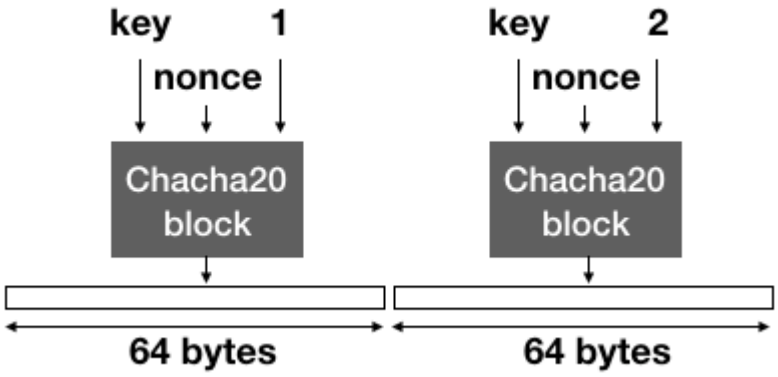


Figure 4.19 ChaCha20's keystream is created by calling an internal block function until enough bytes are produced. One block function call creates 64 bytes of random keystream.

Due to the upperbound on the counter, ChaCha20 can be used to encrypt as many messages as

AES-GCM before its nonce runs into the same issues. Since the output created by this block function is much larger, the size of the messages that can be encrypted is also impacted: you can encrypt 2^{32} x 64 bytes ≈ 274 gigabytes-long messages. If a nonce is used twice to encrypt, similar issues to AES-GCM arise: an observer can obtain the XOR of the two plaintexts by XORing the two ciphertexts, and the authentication key for that nonce can be recovered. These are serious issues that can lead to an attacker being able to forge messages!

NOTE The size of the nonces and the counters are actually not always the same everywhere (both for AES-GCM and ChaCha20-Poly1305), but these are the recommended values from the adopted standards. Still, some cryptographic libraries will accept different size of nonces, and some applications will increase the size of the counter (or the nonce) in order to allow encryption of larger messages (or more messages). Note although that increasing the size of one component, necessarily decrease the size of the other. To prevent this, while allowing a large number of messages to be encrypted under a single key, other standards like XChaCha20-Poly1305 are available. These standards increase the size of the nonce while keeping the rest intact, which is important in cases where the nonce needs to be generated randomly instead of being a counter tracked in the system.

Inside the ChaCha20 block function, a state is formed as shown in figure 4.20.

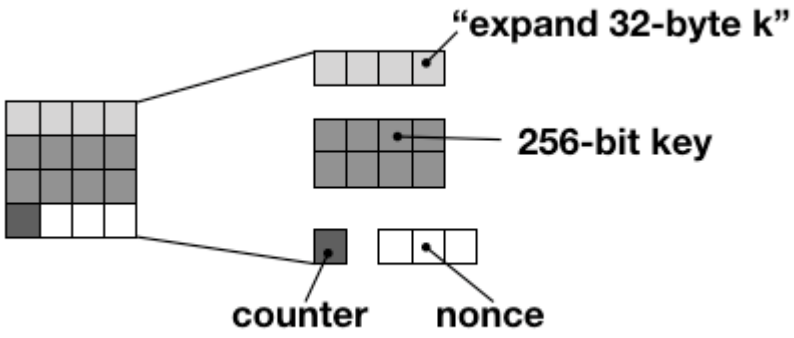


Figure 4.20 The state of ChaCha20 block function. It is formed by 16 words (represented as squares) of 32 bytes each. The first line stores a constant, the second and third lines store the 32-byte symmetric key, the following word stores a 4-byte counter, and the last 3 words store the 12-byte nonce.

This state is then transformed into 64 bytes of keystream by iterating a round function 20 times (hence the 20 in the name of the algorithm). This is very similar to what was done with AES and its round function. The round function is itself calling a **Quarter Round** (QR) function 4 times per round, acting on different words of the internal state every time, depending if the round number is odd or even. See figure 4.21.

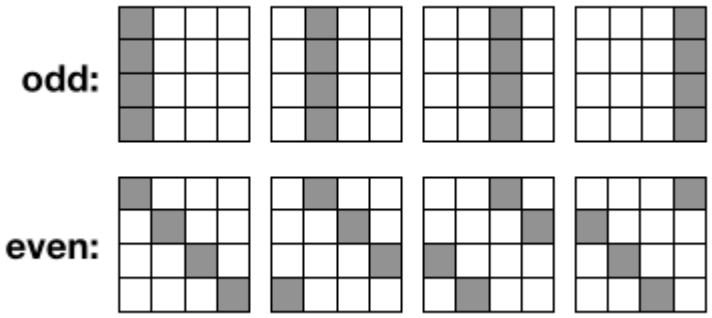


Figure 4.21 A round in ChaCha20 affects all the words contained in a state. As the Quarter Round function only takes 4 arguments, it must be called at least 4 times on different words (greyed in the diagram) to modify all 16 words of the state.

The QR function takes 4 different argument, and updates them using only **A**dd, **R**otate and **X**OR operations. We say that it is an **ARX** stream cipher. This makes ChaCha20 extremely easy to implement and fast in software.

Poly1305 is a MAC created via the Wegman-Carter technique, very much like the GMAC we've previously talked about. See figure 4.22.

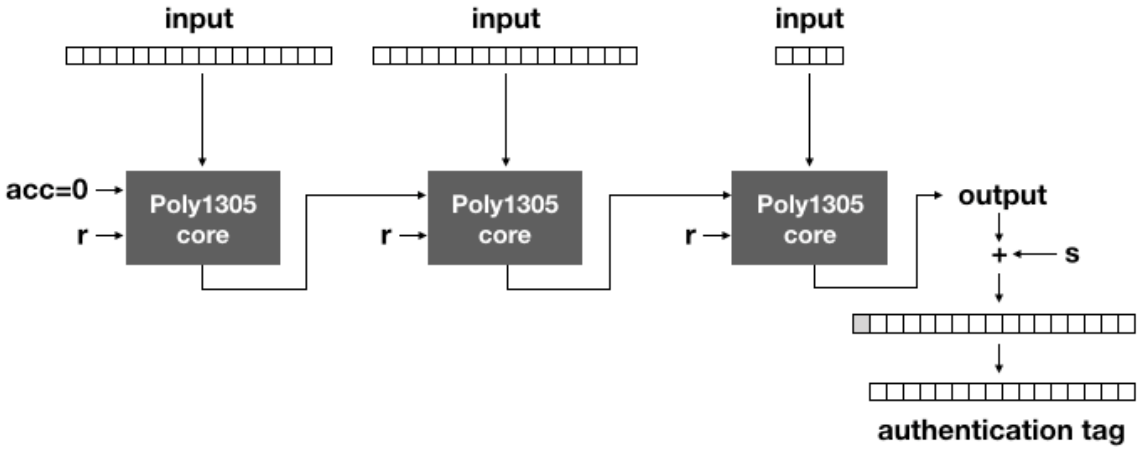


Figure 4.22 Poly1305's core function absorbs an input one block at-a-time by taking an additional accumulator set to 0 initially, and an authentication key `r`. The output is fed as accumulator to the next call of the core function. Eventually the output is added to a random value `s` to become the authentication tag.

Here, `r` can be seen as the authentication key of the scheme, like the authentication key `H` of GMAC. And `s` makes the MAC secure for multiple uses by encrypting the result, thus it must be unique for each usage.

The **Poly1305 core function** mixes the key with the accumulator (set to 0 in the beginning) and the message to authenticate. The operations are simple multiplications modulo a constant `P`.

> **NOTE** Obviously, a lot of details are missing from our description. I seldom mention how to encode data, or how some arguments should be padded before being acted on. These are all implementation specificities that do not matter for us as we are trying to get an intuition of how these things work.

Eventually, we can use ChaCha20 and a counter set to 0 to generate a keystream and derive the 16-byte `r` and 16-byte `s` values we need for Poly1305. I illustrate the resulting AEAD cipher in figure 4.23.

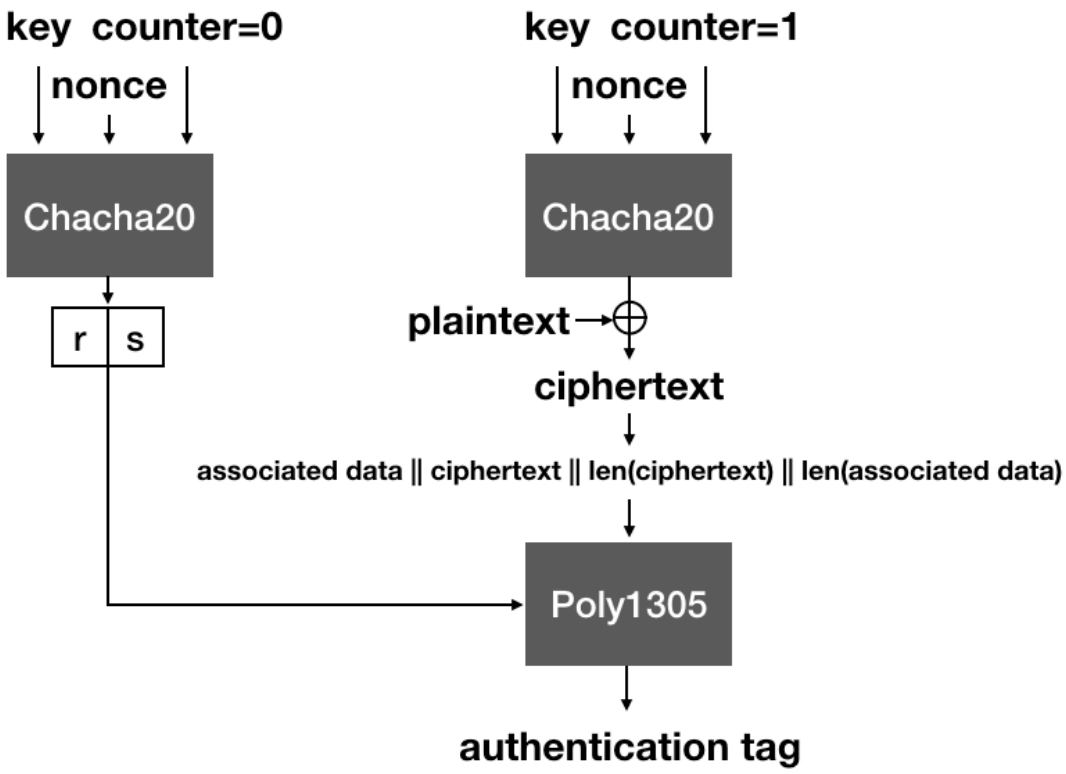


Figure 4.23 ChaCha20-Poly1305 works by using ChaCha20 to encrypt the plaintext, and to derive the keys required by the Poly1305 MAC. Poly1305 is then used to authenticate the ciphertext as well as the associated data.

The normal ChaCha20 algorithm is first used to derive the authentication secrets `r` and `s` needed by Poly1305. The counter is then incremented, and ChaCha20 is used to encrypt a plaintext. After that, the associated data and the ciphertext (and their respective lengths) are passed to Poly1305 to create an authentication tag.

To decrypt, the exact same process is applied. ChaCha20 first verifies the authentication of the ciphertext and the associated data via the tag received. It then decrypts the ciphertext.

4.6 Other kinds of symmetric encryption

Let's pause for a moment and recapitulate the symmetric encryption algorithms you have learned so far:

- **Non-Authenticated Encryption**. It is AES with a mode of operation, but without a MAC. This is insecure in practice as ciphertexts can be tampered with.
- **Authenticated Encryption**. AES-GCM and ChaCha20-Poly1305 being the two most widely adopted ciphers.

The chapter could end here, and it would be fine. Yet, real-world cryptography is not always about the agreed standards, it is also about constraints. Constraints in size, constraints in speed, constraints in format, and so on. To that end, let me give you a brief overview of other types of symmetric encryption that can be useful where AES-GCM and ChaCha20-Poly1305 won't fit.

Key Wrapping. One of the problems of nonce-based AEADs is that they all require a nonce, which takes additional space. Notice that when encrypting a key, you might not necessarily need randomization since what is encrypted is already random and will not repeat with high probabilities (or if it does repeat, it is not a big deal). One well-known standard for key wrapping is the NIST Special Publication 800-38F (Recommendation for Block Cipher Modes of Operation: Methods for Key Wrapping). These key wrapping algorithms do not take an additional nonce or IV, and randomize their encryption based on what they are encrypting. Thanks to this, they do not have to store an additional nonce or IV next to the ciphertexts.

Nonce-Misuse Resistant Authenticated Encryption. In 2006, Rogaway published a new key wrapping algorithm called Synthetic initialization vector (SIV). As part of the proposal, Rogaway notes that the algorithm is not only useful to encrypt keys, but as a general AEAD scheme that is more tolerant to nonce repetitions. As you've learned in this chapter, a repeating nonce in AES-GCM or ChaCha20-Poly1305 can have catastrophic consequences. It not only reveals the XOR of the two plaintexts, but it also allows an attacker to recover an authentication key and to forge valid encryption of messages. The point of a nonce-misuse resistant algorithm is that encrypting two plaintexts with the same nonce will only reveal if the two plaintexts are equal or not, and that's it. It's not great, but it's not as bad as leaking an authentication key obviously. The scheme has gathered a lot of interest and has since been standardized in RFC 8452 (AES-GCM-SIV: Nonce Misuse-Resistant Authenticated Encryption). The trick behind SIV is that the nonce used to encrypt in the AEAD is generated from the plaintext itself, which makes it highly unlikely that two different plaintexts would end up being encrypted under the same nonce.

Disk Encryption. Encrypting the storage of a laptop or a mobile phone has some heavy constraints: it has to be very fast (otherwise the user will notice) and you can only do it in place (saving space is important for a large number of devices). Since the encryption can't expand, AEADs that need a nonce and an authentication tag are no good fit. Instead, unauthenticated encryption is used. To protect against bitflip attacks, large blocks (think thousands of bytes) of

data are encrypted in a way that a single bitflip would scramble the decryption of the whole block. This way, an attack has more chance of crashing the device than accomplishing its goal. These constructions are called wide-block ciphers, although this approach has also been dubbed *poor man's authentication*. Linux systems and some Android devices have adopted this approach using Adiantum, a wide-block construction wrapping the Chacha cipher and standardized by Google in 2019. Still, most devices use non-ideal solutions: both Microsoft and Apple make use of AES-XTS, which is unauthenticated and not wide.

Database Encryption. Encrypting data in a database is tricky. As the whole point is to prevent database breaches from leaking data, the key used to encrypt and decrypt the data must be stored away from the database server. As clients don't have the data itself, they are severely limited in the way they can query the data. The simplest solution is called transparent data encryption (TDE), and simply encrypts selected columns. This works well in some scenarios, although one needs to be careful to authenticate associated data identifying the row and the column being encrypted, otherwise encrypted content can be swapped. Still, one cannot search through encrypted data and so queries have to use the unencrypted columns. Searchable encryption is the field of research that aims at solving this problem. A lot of different schemes have been proposed, but it seems like there is no silver bullet. Different schemes proposes different levels of "searchability" as well as different degradation in security. Blind indexing for example simply allows you to search for exact matches, while order preserving and order revealing encryptions allow you to order results. The bottom line is, the security of these solutions are to be looked at carefuly as they truly are trade-offs.

4.7 Summary

- Symmetric encryption allows two parties to protect the confidentiality of their communications via a shared secret.
- Symmetric encryption needs to be authenticated to be secure. The standard solution is to use an Authenticated Encryption with Associated Data (AEAD) which is an all-in-one authenticated encryption construction.
- AES-GCM and ChaCha20-Poly1305 are the two most widely adopted AEADs, most applications nowadays use either one of them.
- Reusing nonces breaks the authentication of AES-GCM and ChaCha20-Poly1305. Schemes like AES-GCM-SIV are nonce-misuse resistant, while encryption of keys can avoid that problem as nonces are not necessary.
- Real-world Cryptography is about constraints, and AEADs cannot always fit every scenario. This is the case for database or disk encryption, for example, that require the development of new constructions.

5 Key exchanges

This chapter covers

- What key exchanges are and how they can be useful.
- The popular key exchange algorithms: Diffie-Hellman and elliptic curve Diffie-Hellman.
- Security considerations when using key exchanges.

We are now entering the realm of asymmetric cryptography (also called **public-key cryptography**) with our first asymmetric cryptographic primitive: the **key exchange**. A key exchange is, as the name hints, an exchange of key: Alice sends a key to Bob, and Bob sends a key to Alice. This allows the two peers to agree on a shared secret, which can then be used to encrypt communications with an authenticated encryption algorithm, for example.

WARNING As I have hinted in the introduction of this book, there is much more math involved in asymmetric cryptography, therefore the next chapters are going to be a tad more difficult. Don't get discouraged, as what you will learn in this chapter will be helpful to understand many other primitives based on the same fundamentals.

For this chapter you'll need to have read:

- Chapter 3 on message authentication codes.
- Chapter 4 on authenticated encryption.

5.1 What are key exchanges?

Let's start by looking at a scenario where both Alice and Bob want to communicate privately, but have never talked to each other before. This will motivate what key exchanges can unlock in the simplest of situations.

To encrypt communications, Alice can use the authenticated encryption primitive you learned about authenticated in chapter 4. For this, Bob needs to know the same symmetric key, so Alice can generate one and send it over to Bob. After that, they can simply use the key to encrypt their communications. But what if an adversary was **passively** snooping on their conversation? Now they have the symmetric key, and they can decrypt all encrypted content that Alice and Bob are sending to each other.

This is where using a key exchange can be interesting for Alice and Bob. By doing a key exchange, they can obtain a symmetric key that a passive observer won't be able to reproduce.

A key exchange starts with both Alice and Bob **generating some keys**. To do this, they both use some key generation algorithm that generates a key pair: a private key (or secret key) and a public key.

Alice and Bob then sends their respective public keys to each other. "Public" here means that adversaries can observe them without consequences. Alice then uses Bob's public key with her own private key to compute the shared secret. Bob can, similarly, use his private key with Alice's public key to obtain the same shared secret. I illustrate this in figure 5.1.

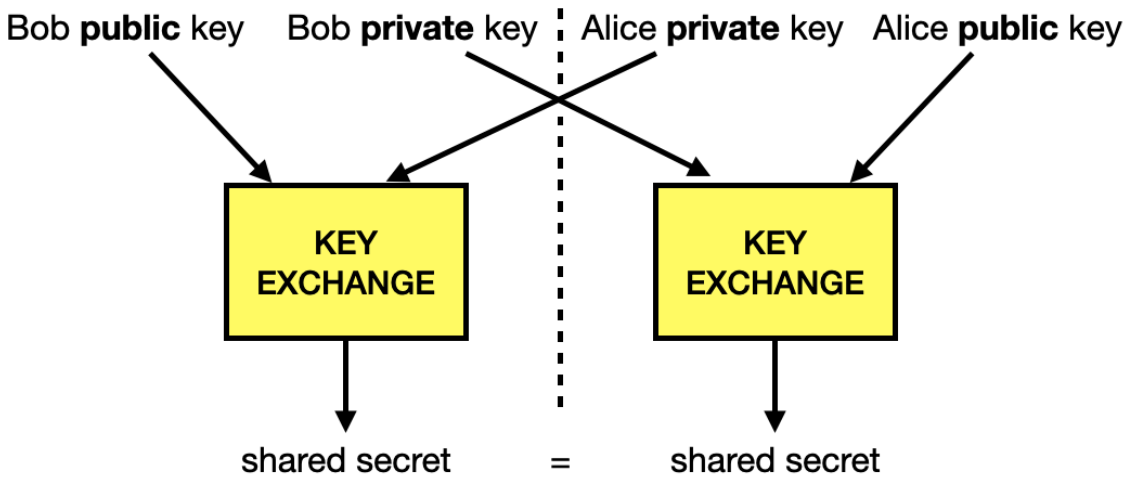


Figure 5.1 A key exchange provides the following interface: it takes your peer's public key and your private key to produce a shared secret. Your peer can obtain the same shared secret by using your public key and their private key.

Knowing how a key exchange works from a high-level, we can now go back to our initial

scenario and see how it helps. By starting their communication with a key exchange, Alice and Bob produce a shared secret to use as a key to an authenticated encryption primitive. Since a MITM adversary observing the exchange cannot derive the same shared secret, they won't be able to decrypt communications. I illustrate this in figure 5.2.

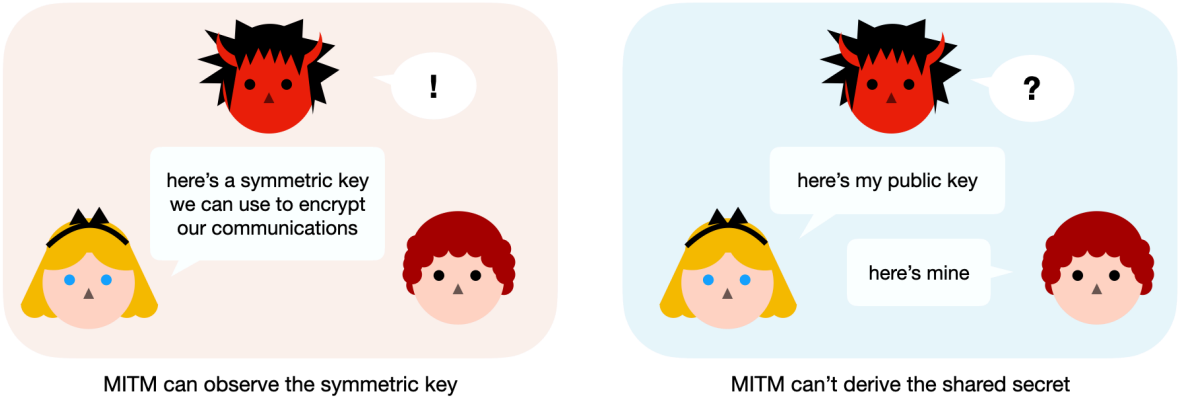


Figure 5.2 A key exchange between two participants allows them to agree on a secret key, while man-in-the-middle adversaries can't derive the same secret key from passively observing the key exchange.

Note that the man in the middle here is passive; an **active** man in the middle would have no problem intercepting the key exchange and impersonating both sides. In this attack, Alice and Bob would effectively perform a key exchange with the MITM, both thinking that they agreed on a key with each other.

The reason this is possible is that none of our characters have a way to verify who the public key they receive really belongs to. The key exchange is **unauthenticated**! I illustrate the attack in figure 5.3.

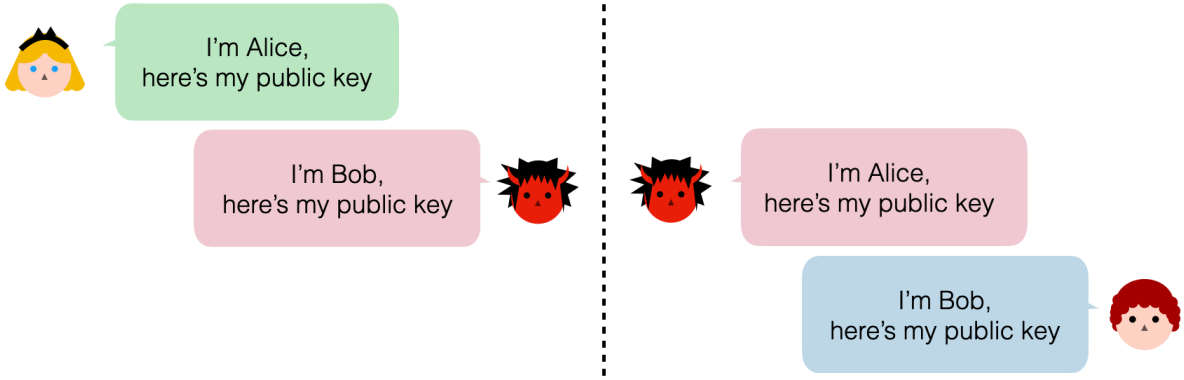


Figure 5.3 An unauthenticated key exchange is vulnerable to an active MITM attacker. Indeed, the attacker can simply impersonate both sides of the connection and perform two separate key exchanges.

Let's take a look at a different scenario to motivate **authenticated key exchanges**. Imagine that you want to run a service that gives you the time of day. Yet, you do not want this information to

be modified by a MITM adversary. Your best bet is to authenticate your responses using the message authentication codes you learned about in chapter 3. Since MACs require a key, you could simply generate one and share it manually with all of your users. This is not great: since every users know the MAC key, they can modify your service's authenticated responses if they ever get to MITM another connection. You could set up a different key per user, but this is not ideal as well: for every new user that wants to connect to your service, you will have to manually provision both your service and the user with a new MAC key. It would be so much better if you didn't have anything to do on the server side, wouldn't it?

Key exchanges can help here! What you could do is have your service generate a key exchange keypair, and provision any new user with the service's public key. This is what an **authenticated key exchange** is: your users know the server's public key, and thus an active MITM adversary cannot impersonate that side of the key exchange. What they can do though, is perform their own key exchange, as the client side of the connection is not authenticated. By the way, when both sides are authenticated, we call that a **mutually-authenticated key exchange**.

This scenario is very common, and the key exchange primitive allows it to scale well with an increase of users. But the scenario doesn't scale well if the number of services increase as well! The internet is a good example: we have many browsers trying to communicate securely with many websites. Imagine if you had to hardcode the public key of all the websites you might one day visit in your browser, and what happens when more websites come online? While key exchanges are useful, they do not scale well in all scenarios without their sister primitive: the **digital signature**. This is just a teaser though, as you will learn about that new cryptographic primitive and how it helps trust to scale in chapter 7.

Key exchanges are rarely used directly in practice, they are often just a building block of a more complex protocol. That being said, they can still be useful on their own sometimes (for example, as we've seen previously against passive adversaries). So let's look at how you would use a key exchange cryptographic primitive in practice.

Libsodium is one of the well-known and widely used C/C++ library. Here is how you would use libsodium in order to perform a key exchange:

Listing 5.1 key_exchange.c

```
unsigned char client_pk[crypto_kx_PUBLICKEYBYTES];   ❶
unsigned char client_sk[crypto_kx_SECRETKEYBYTES];   ❶
crypto_kx_keypair(client_pk, client_sk);   ❶

unsigned char server_pk[crypto_kx_PUBLICKEYBYTES];   ❷
obtain(server_pk);   ❷

unsigned char key_to_decrypt[crypto_kx_SESSIONKEYBYTES];   ❸
unsigned char key_to_encrypt[crypto_kx_SESSIONKEYBYTES];   ❸

if (crypto_kx_client_session_keys(key_to_decrypt, key_to_encrypt,
                  client_pk, client_sk, server_pk) != 0) {   ❹
    abort_session();    ❺
}
```

❶ Use this function to generate the client's keypair (secret key, public key).

❷ We assume at this point that we have some way to obtain the server's public keys.

❸ Instead of generating one shared secret, libsodium goes further and per best practice derives two symmetric keys for us to use: one to decrypt messages from the server, and one to encrypt messages to the server.

❹ We use this function with our secret key and the server's public key in order to perform the key exchange.

❺ If the server's public key is malformed (or even maliciously formed) the function returns an error.

Libsodium hides a lot of details from the developer, while also exposing safe-to-use interfaces. In this instance, libsodium makes use of the **X25519** key exchange algorithm, which you will learn more about later in this chapter.

In the rest of this chapter, you will learn about the different standards used for key exchanges, as well as how they work under the hood.

5.2 The Diffie-Hellman (DH) key exchange

In 1976, Whitfield Diffie and Martin E. Hellman wrote their seminal paper on the Diffie-Hellman (DH) key exchange algorithm entitled "New Direction in Cryptography". What a title! DH was the first key echange algorithm invented, and one of the first formalizations of a public key cryptographic algorithm.

In this section I lay out the math foundations of this algorithm, explain to you how it works, and finally talk about the standards that specify how you can use it in a cryptographic application.

5.2.1 Group theory

The DH key exchange is built on top of a field of mathematics called **group theory**, which is the base of most public-key cryptography today. For this reason, I will spend some time in this chapter giving you the basics on group theory. I will do my best to give good intuitions on how these algorithms work, but there's no way around it: there is going to be some math.

Let's start with the obvious question: what's a **group**?

It's two things:

1. a set of elements
2. a special binary operation (like + or ×) defined on these elements

If the set and the operation managed to satisfy some properties, then we have a group. And if we have a group, then we can do magical things… more on that later.

Note that DH works in a **multiplicative group**: a group where the multiplication is used as the defined binary operation. For this reason, the rest of the explanations uses a multiplicative group as example. I will also often omit the × symbol, for example, I will write $a \times b$ as ab.

I need to be a bit more specific here. For the set and its operation to be a group, they need to have:

- **Closure**, operating on two elements results on another element of the same set. For example, for two elements of the group a and b, a × b results in another group element.
- **Associativity**, operating on several elements at a time can be done in any order. For example, for three elements of the group a, b, c then a(bc) and (ab)c result in the same group element.
- **Identity element**, there exist an element called the identity element. Operating on this identity element does not change the result of the other operand. For example, we can define the identity element as 1 in our multiplicative group. For any group element a, we have a × 1 = a.
- **Inverse element**, there exists an inverse to all group element. For example, for any group element a, there's an inverse element a^{-1} (also written 1/a) such that $a \times a^{-1} = 1$ (also written a × 1/a = 1).

As usual, I illustrate these properties in a more visual way in figure 5.4 to provide you with more material to grasp this new concept.

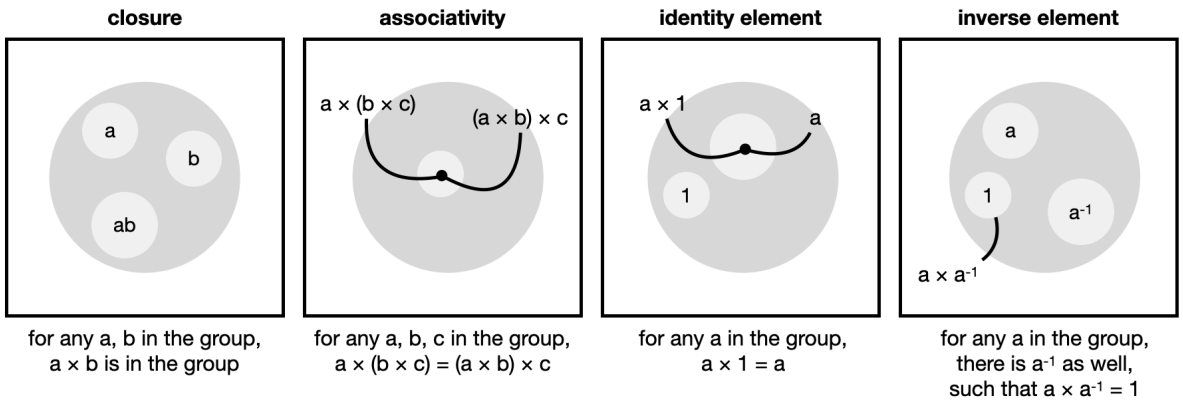


Figure 5.4 The four properties of a group: closure, associativity, identity element, and inverse element.

I can imagine that my explanation of a group can be a bit abstract, so let's see what DH uses as a group in practice.

First, DH uses a group comprised of the set of strictly positive integers 1, 2, 3, 4, …, p-1 where p is a prime number and 1 is the identity element. Different standards will specify different numbers for p, but intuitively it has to be a large prime number for the group to be secure.

> **NOTE** **Remember, a prime number is just a number that can only be divided by 1 or by itself. The first prime numbers are 2, 3, 5, 7, 11, and so on. Prime numbers are everywhere in asymmetric cryptography! And fortunately, we have efficient algorithms to find large ones, but to speed things up most cryptographic libraries will instead look for pseudo-primes (which are numbers that have a high probability of being primes). Interestingly, such optimizations have been broken several times in the past, the most infamous occurence was in 2017 when the ROCA vulnerability found that more than a million devices had generated incorrect primes for their cryptographic applications.**

Second, DH uses the **modular multiplication** as special operation. Before I can explain what a modular multiplication is, I need to explain what **modular arithmetic** is. Modular arithmetic, intuitively, is about numbers that "wrap around" past a certain number called a **modulus**. For example, if we set the modulus to be 5, we say that numbers past 5 go back to 1: 6 becomes 1, 7 becomes 2, and so on. (We also note 5 as 0, but since it is not in our multiplicative group we don't care too much about it.)

The mathematical way to express modular arithmetic, is to take the **remainder** of a number and its **Euclidian division** with the modulus. Let's take, for example the number 7, and write its Euclidian division with 5 as 7 = 5 x 1 + 2. Notice that the remainder is 2. Then we say that 7 = 2 mod 5 (sometimes written 7 ≡ 2 mod 5 as well). The equation above can be read as "*7 is **congruent** to 2 modulo 5*". Similarly:

- 8 = 1 mod 7

- 54 = 2 mod 13
- 170 = 0 mod 17
- and so on…

The classical way of picturing such a concept is with a clock. This is illustrated in figure 5.5.

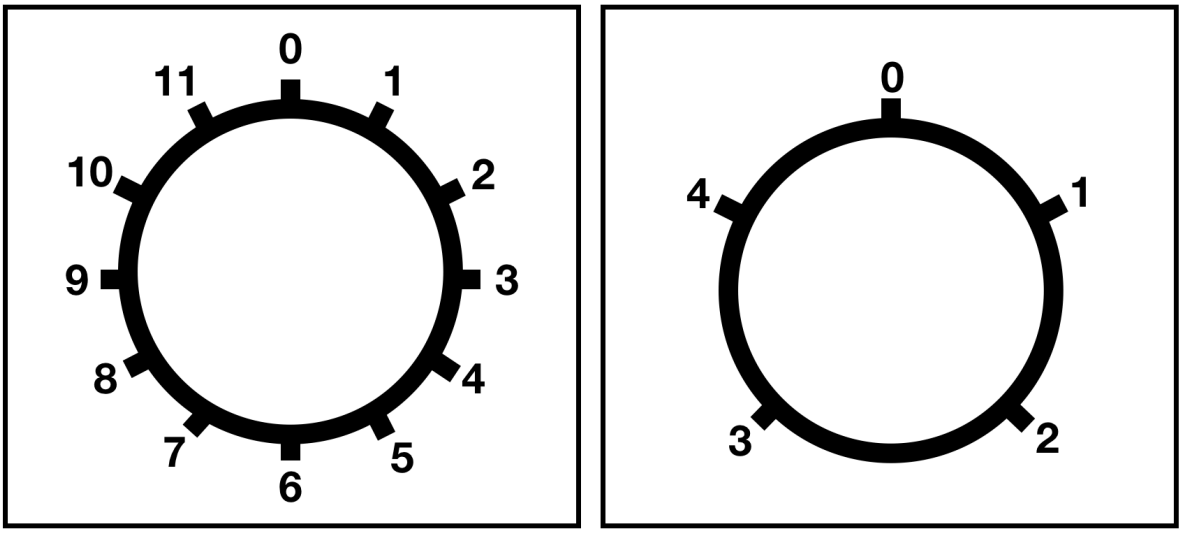


Figure 5.5 The group of integers modulo the prime number 5 can be pictured as a clock that resets to 0 after the number 4. Thus 5 is represented as 0, 6 as 1, 7 as 2, 8 as 3, 9 as 4, 10 as 0, and so on.

A **modular multiplication** is quite natural to define over such a set of numbers. Let's take the following multiplication, for example:

$3 \times 2 = 6$

With what you learned above, you know that 6 is congruent to 1 modulo 5, and thus the equation can be rewritten as

$3 \times 2 = 1 \bmod 5$

Quite straightforward isn't it? Note that the previous equation tells us that 3 is the inverse of 2, and vice versa. So we can also write, for example:

$3^{-1} = 2 \bmod 5$

NOTE	It happens that when we use the positive numbers modulo a prime, only the zero element lacks an inverse (indeed, can you find an element `b` such that 0 x b = 1 mod 5?). This is the reason why we do not include zero as one of our element in our group.

OK, we now have a group: the set of strictly positive integers 1, 2, …, p-1 for p a prime number, along with the modular multiplication. The group we formed also happens to be two things:

- **Commutative**: the order of operations don't matter. For example, given two group elements a and b, then ab = ba. A group that has this property is often called a **Galois group**.
- **A finite field**: a Galois group that has more properties as well as an additional operation (the addition in our example).

For this reason, DH defined over this type of group is sometimes called **finite field Diffie-Hellman (FFDH)**.

If you understand what a group is (and make sure you do before reading any further), then a **subgroup** is just a group contained inside your original group. That is, it's a subset of the group elements, operating on elements of the subgroup will result in another subgroup element, every subgroup element has an inverse in the subgroup, etc.

A **cyclic subgroup** is a subgroup that can be generated from a single **generator** (or **base**). A generator generates a cyclic subgroup by multiplying itself over and over. For example, the generator 4 defines a subgroup consisting of the numbers 1 and 4:

- 4 mod 5 = 4
- 4 x 4 mod 5 = 1
- 4 x 4 x 4 mod 5 = 4 (we start again from the beginning)
- 4 x 4 x 4 x 4 mod 5 = 1
- and so on …

Note that we can also write 4 x 4 x 4 as 4^3.

It happens that when our modulus is prime, every element of our group is a generator of a subgroup. These different subgroups can have different sizes, which we call **orders**. I illustrate this in figure 5.6.

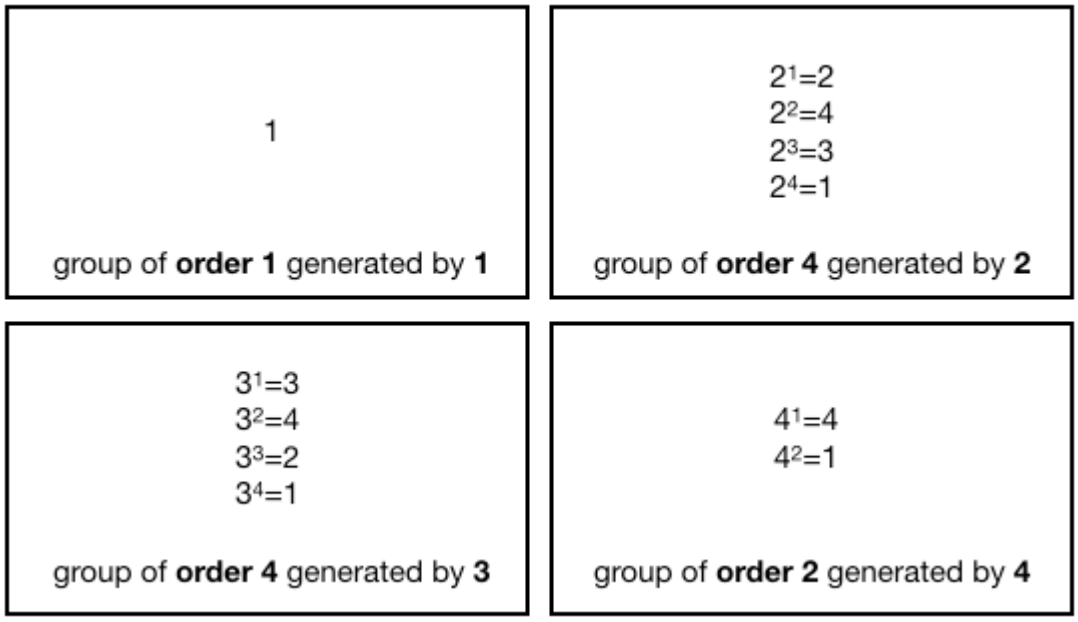


Figure 5.6 The different subgroups of the multiplicative group modulo 5. They all include the number 1 (called the identity element) and have different orders (number of elements).

All right, you now understand:

- A group is a set of numbers with a binary operation that respect some properties (closure, associativity, identity element, inverse element).
- DH works in the Galois group (a group with commutativity) formed by the set of strictly positive numbers up to a prime number (not included) and the modular multiplication.
- In a DH group, every element is a generator of a subgroup.

Groups are the center of a huge amount of different cryptographic primitives. It is thus important to have good intuitions about them, if you want to understand how other cryptographic primitives work.

5.2.2 The discrete logarithm problem, the basis of DH

The security of the DH key exchange relies on the **discrete logarithm problem** in a group, a problem believed to be hard to solve. In this section I briefly introduce this problem.

Imagine that I take a generator, let's say 3, and give you a random element it generates, let's say $2 = 3^x \bmod 5$ for some x unknown to you. Asking you "*what is x?*" is the same as asking you "find the **discrete logarithm of 2 in base 3**." Thus, the discrete logarithm problem in our group is about finding out how many times we multiplied the generator with itself in order to produce a given group element. This is an important concept; take a few minutes to think about it before continuing.

In our example group, you can quickly find out that 3 is the answer. Indeed $3^3 = 2 \bmod 5$. But if we picked a way bigger prime number than 5, things get much more complicated: it becomes

hard to solve. This is the secret sauce behind Diffie-Hellman!

You now know enough to understand how to generate a keypair in DH:

1. All the participants must agree on a **large prime *p*** and a **generator *g***.
2. Each participant generates a random number x, this becomes their **private key**.
3. Each participant derives their **public key** as $g^x \bmod p$.

The fact that the discrete logarithm problem is hard means that no one should be able to recover the private key out of the public key. I illustrate this in figure 5.7.

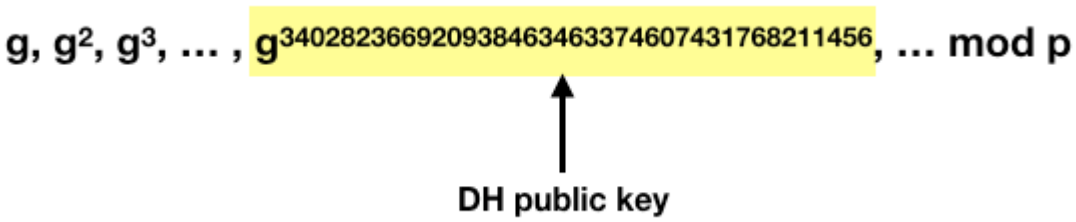


Figure 5.7 Choosing a private key in Diffie-Hellman is like choosing an index in a list of numbers produced by a generator g. The discrete logarithm problem is to find the index from the number alone.

While we do have algorithms to compute discrete logarithms, they are not efficient in practice. On the other hand, if I give you the solution x to the problem , you have extremely efficient algorithms at your disposition to check that, indeed, I provided you with the right solution $g^x \bmod p$. If you are interested, the state-of-the-art technique to compute the modular exponentiation is called **square and multiply**, and computes the result efficiently by going through x bit by bit .

> **NOTE** Like everything in cryptography, it is not impossible to find a solution by simply trying to guess. Yet, by choosing large enough parameters (here a large prime number), it is possible to reduce the efficacy of such a search for a solution down to negligible odds. Meaning that even after hundreds of years of random tries, your odds of finding a solution should still be statistically close to zero.

Great. How do we use all of this math for our DH key exchange algorithm?

Imagine that:

- Alice has a private key a and a public key $A = g^a \bmod p$.
- Bob has a private key b and a public key $B = g^b \bmod p$.

With the knowledge of Bob's public key, Alice can compute the shared secret as

$B^a \bmod p$

Bob can do a similar computation with Alice's public key and his own private key

$A^b \bmod p$

Very naturally, we can see that these two calculations end up computing the same number!

$B^a = (g^b)^a = g^{ab} = (g^a)^b = A^b \bmod p$

And that's the magic of DH. From an outsider, just observing the public keys A and B does not help in any way to compute the result of the key exchange $g^{ab} \bmod p$.

NOTE **By the way, in theoretical cryptography the idea that observing $g^a \bmod p$ and $g^b \bmod p$ does not help you to compute $g^{ab} \bmod p$ is called the Computational Diffie-Hellman Assumption (CDH). It is often confused with the stronger Decisional Diffie-Hellman Assumption (DDH) which intuitively states that given the tuple ($g^a \bmod p$, $g^b \bmod p$, $z \bmod p$), nobody should be able to confidently guess if the latter element is the result of a key exchange between the two public keys ($g^{ab} \bmod p$) or just a random element of the group. Both are very useful theoretical assumptions that have been used to build many different algorithms in cryptography.**

Next, you will learn about how real-world applications make use of this algorithm, and the different standards that exist.

5.2.3 Diffie-Hellman standards

Now that you have seen how Diffie-Hellman works, you understand that participants need to agree on a set of parameters, specifically on a prime number p and a generator g.

In this section you'll learn about how real-world applications choose these parameters, and what are the different standards that exist.

First thing first, the prime number p. As I stated earlier, the bigger, the better. Since DH is based on the discrete logarithm problem, its security is directly correlated to the best attacks known on the problem. Any advances in this area can weaken the algorithm. With time, we've managed to obtain a pretty good idea of how fast (or slow) these advances are, and how much is enough security. The current known best practices are to use a prime number of **2048 bits**.

> **NOTE** In general, https://keylength.com summarizes recommendations on parameter lengths for common cryptographic algorithms. The results are taken from authoritative documents produced by research groups or government bodies like the ANSSI (France), the NIST (US), and the BSI (Germany). While they do not always agree, they often converge towards similar orders of magnitude.

In the past, many libraries and software were generating and hardcoding their own parameters. Unfortunately, they were often found to be either weak, or worse, backdoored. While blindly following standards might seem like a good idea in general, DH is one of the unfortunate counter-examples that exist, as it was found that some of them might have been backdoored (RFC 5114 was found specifying broken DH groups by Antonio Sanso in 2016). Due to all of these issues, newer protocols and libraries have converged towards either deprecating DH (in favor of ECDH) or using the groups defined in the better standard RFC 7919.

For this reason, best practice nowadays is to use RFC 7919 which defines several groups of different sizes and security. For example, **ffdhe2048** is the group defined by the 2048-bit prime modulus

p =
323170060713110073001535134778251633624880571334890751745884341392698068341362100027920563626401646854585563579353308169288290230805734726252735547424612457410262025279165729728627063003252634282131457669314142236542209411113486299916574782680342305530863490506355577122191878903327295696961297438562417412362372251973464026918557977679768230146253979330580152268587307611975324364674758554607150438968449403661304976978128542959586595975670512838521327844685229255045682728791137200989318739591433741758378260002780349731985520606075332341226032546840881200311059074842810039949669561196969562486290323380728391270391

and with generator g = 2

> **NOTE** It is common to choose the number 2 for the generator, as computers are quite efficient at multiplying with 2 (it's a simple left shift <<instruction).

The group size (or order) is also specified as q = (p-1)/2.

This implies that both private keys and public keys will be around 2048-bit of size.

In practice, these are quite large sizes for keys (compare that with symmetric keys for example, that are usually 128-bit). You will see in the next section that defining a group over the elliptic curves allow us to obtain much smaller keys for the same amount of security.

5.3 The elliptic curve Diffie-Hellman (ECDH) key exchange

It turns out that the Diffie-Hellman algorithm, which we just discussed, can be implemented in different types of groups, not just the multiplicative groups modulo a prime number. It also turns out that a group can be made from elliptic curves, a type of curves studied in mathematics. The idea was proposed in 1985 by Neal Koblitz and Victor S. Miller independently, and much later (in 2000) adopted when cryptographic algorithms based on elliptic curves started seeing standardization. The world of applied cryptography quickly adopted elliptic curve cryptography as it provided much smaller keys than the previous generation of public key cryptography. Compared to the recommended 2048-bit parameters in DH, parameters of 256 bits were possible with the elliptic curve variant of the algorithm.

5.3.1 What's an elliptic curve?

Let's now explain how elliptic curves work. First and foremost, it is good to understand that elliptic curves are just curves! Meaning that they are defined by all the coordinates x and y that solves an equation. Specifically this equation

$$y^2 + a_1 x y + a_3 y = x^3 + a_2 x^2 + a_4 x + a_6$$

for some a_1, a_2, a_3, a_4, and a_6.

NOTE **Note that for most practical curves today, this equation can be simplified as the short Weierstrass equation $y^2 = x^3 + ax + b$ (where $4a^3 + 27b^2 \neq 0$). While the simplification is not possible for two types of curves (called binary curves and curves of characteristic 3), they are used rarely enough that we will use the Weierstrass form in the rest of this chapter.**

Figure 5.8 shows an example of an elliptic curve, with two points taken at random.

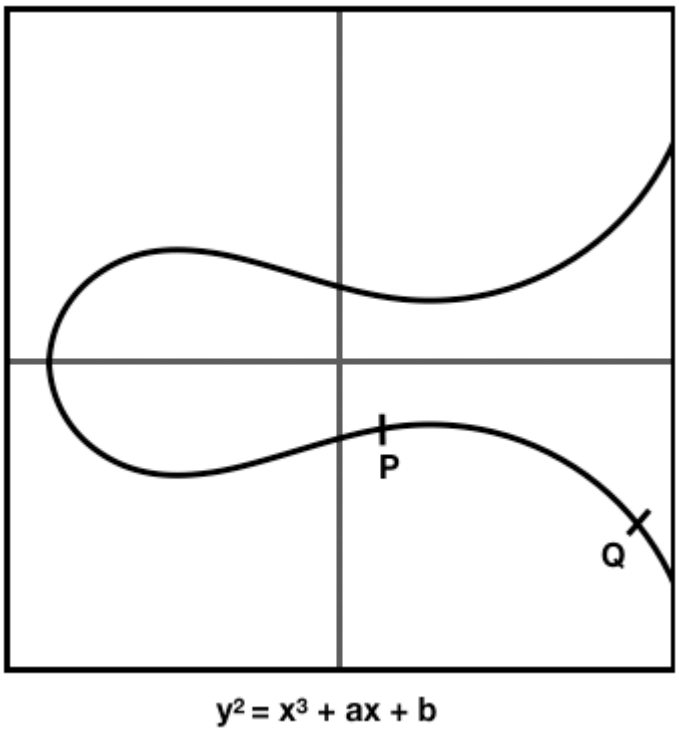


Figure 5.8 One example of an elliptic curve defined by an equation.

At some point in the history of elliptic curves, it was found that a **group** could be constructed over them. From there, implementing Diffie-Hellman on top of these groups was straightforward. I will use this section to explain the intuition behind elliptic curve cryptography.

Groups over elliptic curves are often defined as **additive groups**. Unlike multiplicative groups defined in the previous section, the + sign is used instead.

> **NOTE** Using an addition or a multiplication does not matter much in practice, it is just a matter of preference. While most of cryptography uses a multiplicative notation, the literature around elliptic curves has gravitated around an additive notation, and thus this is what I will use when referring to elliptic curve groups in this book.

This time, I will define the operation before defining the elements of the group. Our **addition operation** is defined in the following way:

1. Draw a line going through two points you want to add. The line hits the curve at another point.
2. Draw a vertical line from this newly found point. The vertical line hits the curve in yet another point.
3. This point is the result of adding the original two points together.

This process is illustrated in figure 5.9.

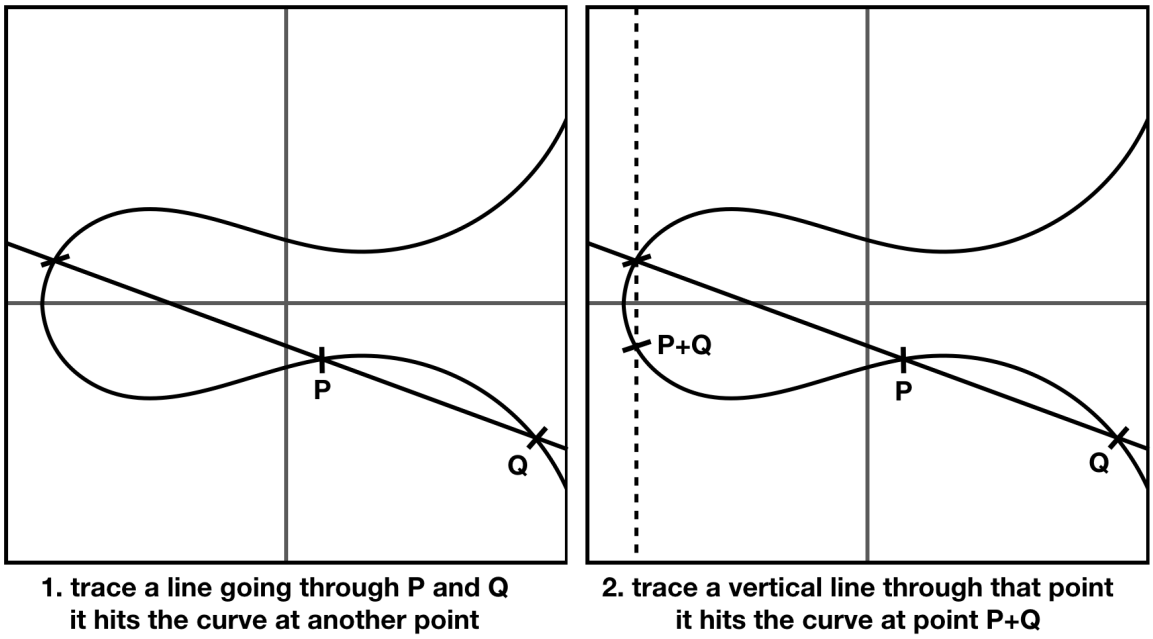


Figure 5.9 An addition operation can be defined over points of an elliptic curve by using geometry!

There are two special cases where this rule won't work. Let's define them as well:

- How do we add a point to itself? The answer is to draw the tangent to that point (instead of drawing a line between two points).
- What happens if the line we draw in step 1 (or step 2) does not hit the curve at any other point? Well, this is embarassing and we need this special case to work and produce a result. The solution is to define the result as a made-up point (something we made up). This newly-invented point is called the **point at infinity** (that we usually write with a big letter O).

Figure 5.10 illustrates these special cases.

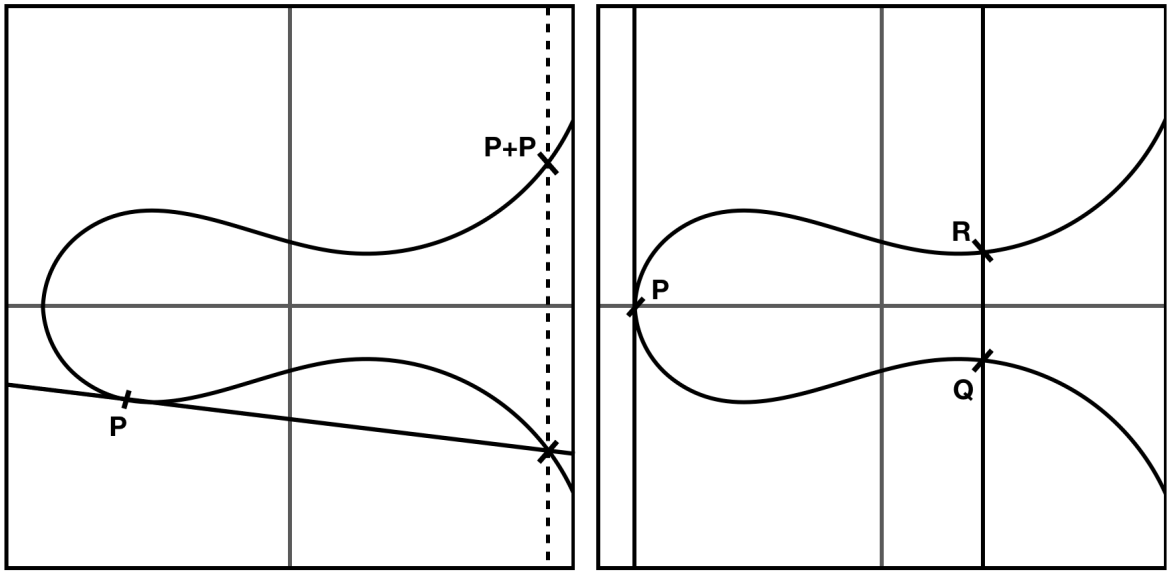


Figure 5.10 Building on figure 15.5, addition on an elliptic curve is also defined when adding a point to itself, or when two points cancel each other to result in the point at infinity.

I know this point at infinity is some next-level weirdness, but don't worry too much about it. It is really just something we came up with in order to make the addition operation work. Oh and by the way, it behaves like a zero, and it is our **identity** element:

$$O + O = O$$

and for any point P on the curve

$$P + O = P$$

All good. So far we've seen that to create a group on top of an elliptic curve, we need:

- An elliptic curve equation that defines a set of valid points.
- A definition of what an addition means in this set.
- An imaginary point called a point at infinity.

I know this is a lot of information to unpack, but we are missing one last thing. Elliptic curve cryptography is defined over a finite field. In practice, what this means is that our coordinates are the numbers 1, 2, …, p-1 for some large prime number p. This should sound familiar!

For this reason, when thinking of elliptic curve cryptography, you should be thinking of a graph that looks much more like the one on the right in figure 5.11.

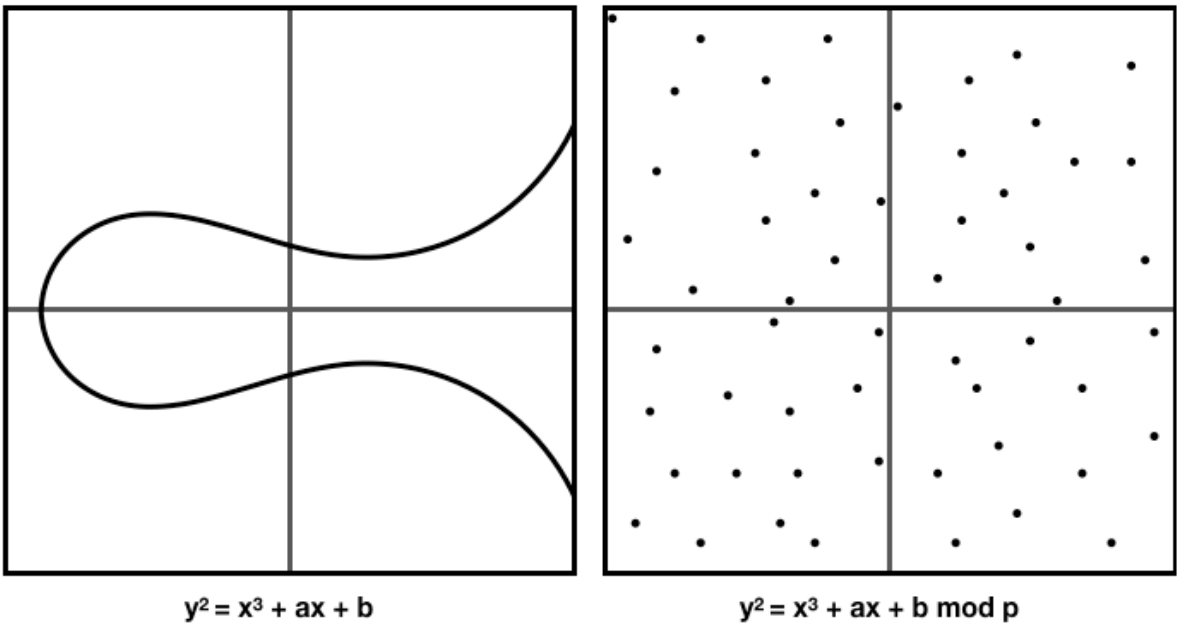


Figure 5.11 Elliptic Curve Cryptography (ECC) in practice is mostly specified with elliptic curves in coordinates modulo a large prime number p. This means that what we use in cryptography looks much more like the right graph than the left graph.

And that's it!

We now have a group we can do cryptography on. The same way we had a group made with the numbers (excluding 0) modulo a prime number and a multiplication operation for Diffie-Hellman.

How can we do Diffie-Hellman with this group defined on elliptic curves? Let's see how the **discrete logarithm** works now in this group.

Let's take a point G and add it to itself x times to produce another point P via the addition operation we defined. We can write that as $P = G+...+G$ (x times), or use some mathematical syntactic sugar to write that as $P = [x]G$ which reads "x times G". The **elliptic curve discrete logarithm problem (ECDLP)** is to find the number x from knowing just P and G.

> **NOTE** we call [x]G a scalar multiplication, as x is usually called a scalar in such groups.

5.3.2 How does the elliptic curve Diffie-Hellman key exchange work?

That's all we needed, you now know enough to understand how to generate a keypair in ECDH:

1. All the participants agree on an **elliptic curve equation**, a **finite field** (most likely a prime number), and a **generator G** (usually called **base point** in elliptic curve cryptography).

2. Each participant generates a random number X, this becomes their **private key**.
3. Each participant derives their **public key** as [X]G.

As the elliptic curve discrete logarithm problem is hard, you guessed it, no one should be able to recover your private key just by looking at your public key. I illustrate this in figure 5.12.

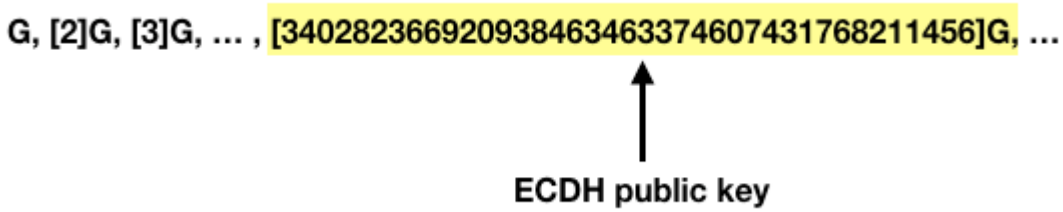


Figure 5.12 Choosing a private key in ECDH is like choosing an index in a list of numbers produced by a generator (or base point) G. The elliptic curve discrete logarithm problem (ECDLP) is to find the index from the number alone.

All of this might be a bit confusing, as the operation we had defined for our DH group was a multiplication, and for an elliptic curve we now use an addition. Again, these distinctions do not matter at all, as they are equivalent. You can see a comparison in figure 5.13.

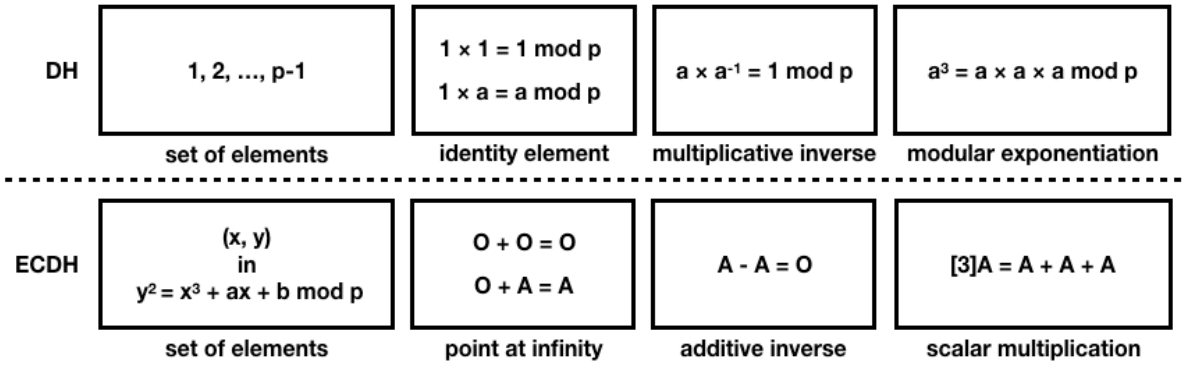


Figure 5.13 Some comparisons between the group used in Diffie-Hellman and the group used in Elliptic Curve Diffie-Hellman.

You should now be convinced that the only thing that matters for cryptography is that we have a group defined with its operation, and that the discrete logarithm for this group is hard. See figure 5.14.

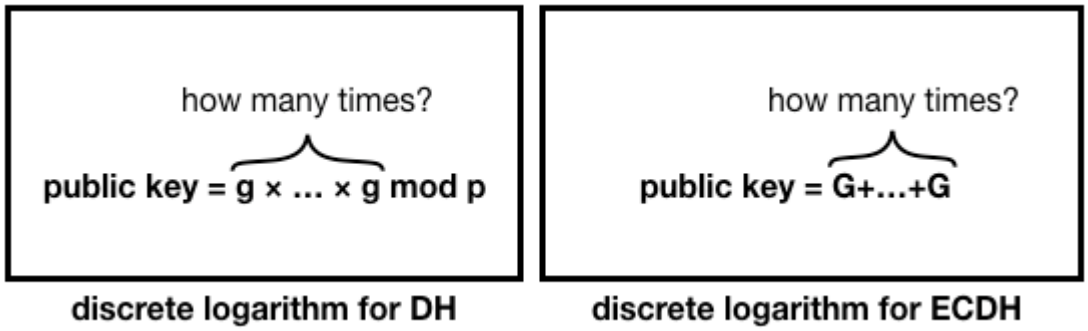


Figure 5.14 A comparison between the discrete logarithm problem modulo large primes, and the discrete logarithm problem in elliptic curve cryptography. They both relate to the Diffie-Hellman key exchange as the problem is to find the private key from a public key.

Last note on the theory, the group we formed on top of elliptic curves has some difference with the group we formed with the strictly positive integers modulo a prime number. Due to some of these differences, the strongest attacks known against DH (index calculus) do not work well on the elliptic curve groups. This is the main reason why parameters for ECDH can be much much lower than the parameters for DH, for the same level of security.

OK we are done with the theory. Let's go back to defining ECDH. Imagine that:

- Alice has a private key a and a public key A = [a]G.
- Bob has a private key b and a public key B = [b]G.

With the knowledge of Bob's public key, Alice can compute the shared secret as

[a]B

Bob can do a similar computation with Alice's public key and his own private key

[b]A

Very naturally, we can see that these two calculations end up computing the same number!

[a]B = [a][b]G = [ab]G = [b][a]G = [b]A

And no passive adversary should be able to derive the shared point just from observing the public keys.

Looks familiar right?

Next, let's talk about standards!

5.3.3 Elliptic curve Diffie-Hellman standards

Elliptic curve cryptography has remained at its full strength since it was first presented in 1985. […] The United States, the UK, Canada and certain other NATO nations have all adopted some form of elliptic curve cryptography for future systems to protect classified information throughout and between their governments

– NSA The Case for Elliptic Curve Cryptography (2005)

The standardization of ECDH has been pretty chaotic. Many standardization bodies have come out and specified a large number of different curves, which was then followed by many flamewars over which curve was more secure or more efficient. A large amount of research, mostly led by Daniel J. Bernstein, pointed out the fact that a number of curves standardized by the NIST could potentially be part of a weaker class of curves only known to the NSA.

I no longer trust the constants. I believe the NSA has manipulated them through their relationships with industry.

– Bruce Schneier The NSA Is Breaking Most Encryption on the Internet (2013)

Nowadays, most of the curves in use come from a couple standards, and most applications have fixated on two curves: **P-256** and **Curve25519**. In the rest of this section I will go over these curves.

NIST FIPS 186-4 (Digital Signature Standard), initially published as a standard for signatures in 2000, contains an appendix specifying 15 curves for use in ECDH. One of these curves, **P-256**, is the most widely used curve on the internet. The curve was also specified in SEC 2 version 2 (Recommended Elliptic Curve Domain Parameters), published in 2010 under a different name: **secp256r1**.

P-256 is defined with the short short Weierstrass equation:

$y^2 = x^3 + ax + b \bmod p$

where a = -3 and

b = 41058363725152142129326129780047268409114441015993725554835256314039467401291

and

$p = 2^{256} – 2^{224} + 2^{192} + 2^{96} – 1$

It defines a curve of prime order

n = 115792089210356248762697446949407573529996955224135760342422259061068512044369

meaning that there are exactly n points on the curve (including the point at infinity).

The base point is specified as

G = (4843956129390645175905258525279791420276294952604174799584408071708240463528 6, 3613425095674979579858512791958788195661110667298501507187719825356841440510 9)

The curve provides 128 bits of security. For applications that are using other cryptographic algorithms providing 256-bit security instead of 128 bits of security (for example AES with a 256-bit key), P-521 is available in the same standard to match the level of security.

> **NOTE** **Interestingly P-256, and other curves defined in FIPS 186-4, are said to be generated from a seed. For P-256, the seed is known to be the bytestring `0xc49d360886e704936a6678e1139d26b7819f7e90`. I've talked about this notion of "nothing-up-my-sleeve" numbers before, constants that aim at proving that there was no room for backdooring the design of the algorithm. Unfortunately, there isn't much explanation behind the P-256 seed other than the fact that it is specified along the curve's parameter.**

RFC 7748 (Elliptic Curves for Security), which was published in 2016, specifies two curves: **Curve25519** and **Curve448**. Curve25519 offers approximately 128 bits of security, while Curve448 offers around 224 bits of security for protocols that wish to hedge against potential advances in the state of attacks against elliptic curves.

I will only talk about Curve25519 here, which is a Montgomery curve defined by the equation

$y^2 = x^3+486662x^2+x \bmod p$

where

$p = 2^{255} - 19$

Curve25519 has an order

$n = 2^{252} + 27742317777372353535851937790883648493$

and the base point used is

G = (9 , 14781619447589544791020593568409986887264606134616475288964881837755586237401)

The combination of ECDH with Curve25519 is often dubbed **X25519**.

5.4 Small subgroup attacks and other security considerations

Today, **I would advise you to use ECDH over DH** due to the size of the keys, the lack of known strong attacks, the quality of the available implementations, and the fact that elliptic curves are fixed and well-standardized as opposed to DH groups which are all over the place. The latter point is quite important: using DH means potentially using some broken standard (RFC 5114), using protocols that make use of random groups (for example, in TLS the server gets to choose any modulus they want), using software that uses broken custom DH groups (in 2016, the socat network tool was found to use a malicious-looking DH group), and so on.

If you do have to use Diffie-Hellman, make sure to **stick to the standards**. The standards I mentioned previously make use of **safe primes** as modulus: primes of the form $p = 2q + 1$ where q is another prime number. The point is that groups of this form only have two subgroups: a small one of size 2 (generated by -1) and a large one of size q. (This is the best you can get by the way: there exist no prime order groups in DH.) The scarcity of small subgroups prevent a type of attack known as **small subgroup attack** (more on that later). Safe primes create such secure groups because of two things:

1. The order of a multiplicative group modulo a prime p is calculated as p - 1.
2. The orders of a group's subgroups are the factors of the group's order (this is Lagrange theorem).

Hence, the order of our multiplicative group modulo a safe prime is $p - 1 = (2q + 1) - 1 = 2q$, which has factors 2 and q, which means that its subgroups can only be of order 2 or q. In such groups, small subgroup attacks are not possible as they require many small subgroups.

A small subgroup attack is an attack on key exchanges, in which an attacker sends several invalid public keys to leak bits of your private key gradually. The invalid public keys are generators of small subgroups. For example, an attacker could choose -1, the generator of a subgroup of size 2, as public key and send it to you. By doing your part of the key exchange, the resulting shared secret is an element of the small subgroup (-1 or 1). This is because you just raised the small subgroup generator (the attacker's public key) to the power of your private key. Depending on what you do with that shared secret, the attacker could guess what it is, and leak some information about your private key. With our example of malicious public key, if your private key was even the shared secret would be 1, and if your private key was odd the shared secret would be -1. As a result, the attacker learned one bit of information: the least significant bit of your private key. Many subgroups of different sizes can lead to more opportunities for the attacker to learn more about your private key, until the entire key is recovered. I illustrate this issue in figure 5.15.

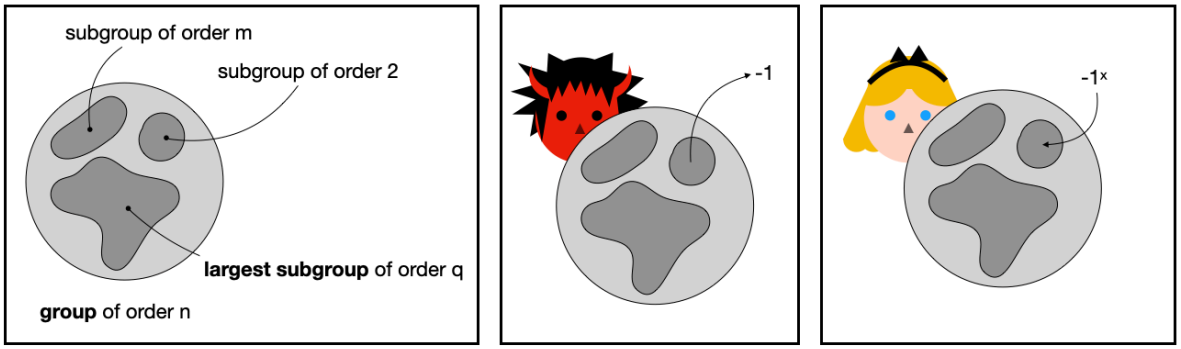


Figure 5.15 A small subgroup attack impacts DH groups that have many subgroups. By choosing generators of small subgroups as public key, an attacker can leak bits of the private key little by little.

While it is always a good idea to verify if the public key you received is in the correct subgroup, not all implementations do it. In 2016, a group of researchers analyzed 20 different DH implementations and found that none was validating public keys (see "Measuring small subgroup attacks against Diffie-Hellman"). So make sure that DH implementations you're using do. You can do this by raising the public key to the order of the subgroup, which should give you back the identity element if it is an element of that subgroup.

On the other hand, elliptic curves allow for groups of prime order! That is, they have no small subgroups (besides the subgroup of size 1 generated by the identity element), and thus they are secure against small subgroup attacks. Well, not so fast… It was found in 2000 by Biehl, Meyer and Muller that small subgroup attacks are possible even in such prime-order elliptic curve groups due to an attack called **invalid curve attack**.

The idea behind invalid curve attacks is the following. First, the formulas to implement scalar multiplication for elliptic curves that use the short Weierstrass equation $y^2 = x^3 + ax + b$ (like NIST's P-256) are independent of the variable b. This means that an attacker can find different curves with the same equation except for the value b, and some of these curves will have many small subgroups… You probably know where this is going: the attacker chooses a point in another curve that exhibits small subgroups, and sends it to a targetted server. The server goes on with the key exchange by performing a scalar multiplication with the given point, effectively doing a key exchange on a different curve. This trick ends up reenabling the small subgroup attack, even on prime-order curves.

The obvious way to fix this is to, again, validate public keys. This can be done easily by checking if the public key is not the point at infinity, and by plugging the received coordinates into the curve equation to see if it solves it. Unfortunately, it was shown in 2015 by Jager, Schwenk, and Somorovsky that several popular implementations did not perform these checks (see "Practical Invalid Curve Attacks on TLS-ECDH").

If using ECDH, I would advise you to use the X25519 key exchange due to the quality of the design (which takes into account invalid curve attacks), the quality of available implementations, and the resistance against timing attacks by design.

Curve25519 has one caviat though: it is not a prime-order group… The curve has two subgroups: a small subgroup of size 8 and a large subgroup used for ECDH. On top of that, the original design did not prescribe validating received points, and implementations in turn did not implement these checks. This led to issues being found in different types of protocols that were making use of the primitive in more exotic ways. (One of them I found in the Matrix messaging protocol, which I talk about in chapter 11.)

Not verifying public keys can have unexpected behaviors with X25519, the reason is that the key exchange algorithm does not have **contributory behavior**: it does not allow both parties to *contribute* to the final result of the key exchange. Specifically, one of the participant can force the outcome of the key exchange to be all zeros by sending a point in the small subgroup as public key.

RFC 7748 does mention this issue, and proposes to check that the resulting shared secret is not the all-zero output, yet lets the implementor decide to do the check or not. I would recommend making sure that your implementation performs the check, although you're likely not going to run into any issues unless you use X25519 in a non-standard way.

Since many protocols rely on Curve25519, this has been an issue for more than just key exchanges. **Ristretto**, the Internet Draft (soon-to-be RFC) https://tools.ietf.org/html/draft-hdevalence-cfrg-ristretto-01, is a construction that adds an extra layer of encoding to Curve25519, effectively simulating a curve of prime order. The construction has been gaining traction, as it simplifies the security assumptions made by other types of cryptographic primitives that want to benefit from Curve25519 but want a prime-order field.

5.5 Summary

- Unauthenticated key exchanges allow two parties to agree on a shared secret, while preventing any passive MITM from being able to derive it as well.
- An authenticated key exchange prevents an active MITM from impersonating one side of the connection, while a mutually-authenticated key exchange prevents an active MITM from impersonating both sides.
- One can perform an authenticated key exchange by knowing the other party's public key, this doesn't always scale and signatures will unlock more complex scenarios (see chapter 7).
- Diffie-Hellman (DH) is the first key exchange algorithm invented, and is still widely used.
- The recommended standard to use for DH is RFC 7919 which includes several parameters to choose from. The smallest option is the recommended 2048-bit prime parameter.
- Elliptic Curve Diffie-Hellman (ECDH) has much smaller key sizes than DH. For 128 bits of security, DH needs 2048-bit parameters whereas ECDH needs 256-bit parameters.
- The most widely used curves for ECDH are P-256 and Curve25519. Both provide 128 bits of security. For 256-bit of security, P-521 and Curve448 are available in the same standards.
- Make sure that implementations verify the validity of public keys they receive, as they are the source of many bugs.

6 Asymmetric encryption and hybrid encryption

This chapter covers

- Asymmetric Encryption can be used to encrypt secrets to a public key.
- Hybrid Encryption can be used to encrypt large amounts of data to a public key.
- The standards for Asymmetric and Hybrid Encryption.

You've learned about authenticated encryption in chapter 4, which is a form of symmetric encryption. Authenticated encryption allowed you to encrypt data to someone else who shared the same symmetric key. This is an extremely useful cryptographic primitive, yet in the real-world, there exist many situations where different peers do not have a shared secret. chapter 5 introduced asymmetric cryptography and how key exchanges allow two participants who are aware of each other's public key to derive a shared secret in the open. This chapter bridges asymmetric cryptography with symmetric cryptography, showing you how you can encrypt to a person with whom you do not share a secret yet, as long as you know their public key.

For this chapter you'll need to have read:

- Chapter 4 on authenticated encryption.
- Chapter 5 on key exchanges.

Let's get started!

6.1 What is asymmetric encryption?

The first step to understanding how to encrypt a message to someone is **asymmetric encryption** (also called **public-key encryption**). In this section you will learn about this cryptographic primitive and its properties.

Let's take a look at the following real-world scenario: **encrypted emails**.

You probably know that all the emails you send are sent in the clear, for anyone (sitting in between your and your recipient's email providers) to read.

That's not great. How do you fix this? You could use a cryptographic primitive like AES-GCM, which you've learned about in chapter 4 on authenticated encryption. To do that, you would need to set up a different shared symmetric secret for each person that wants to message you (using the same shared secret with everyone would be very bad, can you see why?) In practice, how can you do this? You can't expect to know in advance who'll want to message you.

One solution is **asymmetric encryption**! With such a primitive, anyone can encrypt messages to you using your **public key**, and you are the only one who can decrypt such messages via your associated **private key**. See figure 6.1.

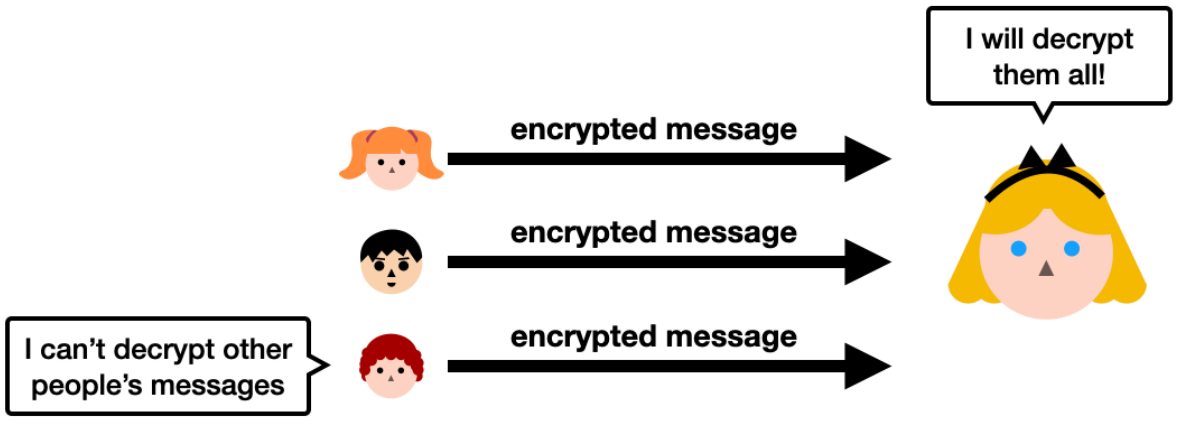


Figure 6.1 With asymmetric encryption, anyone can use Alice's public key to send her encrypted messages. Only Alice, who owns the associated private key, can decrypt these messages.

To set up such an algorithm, you first need to **generate a key pair** via some algorithm as illustrated in figure 6.2.

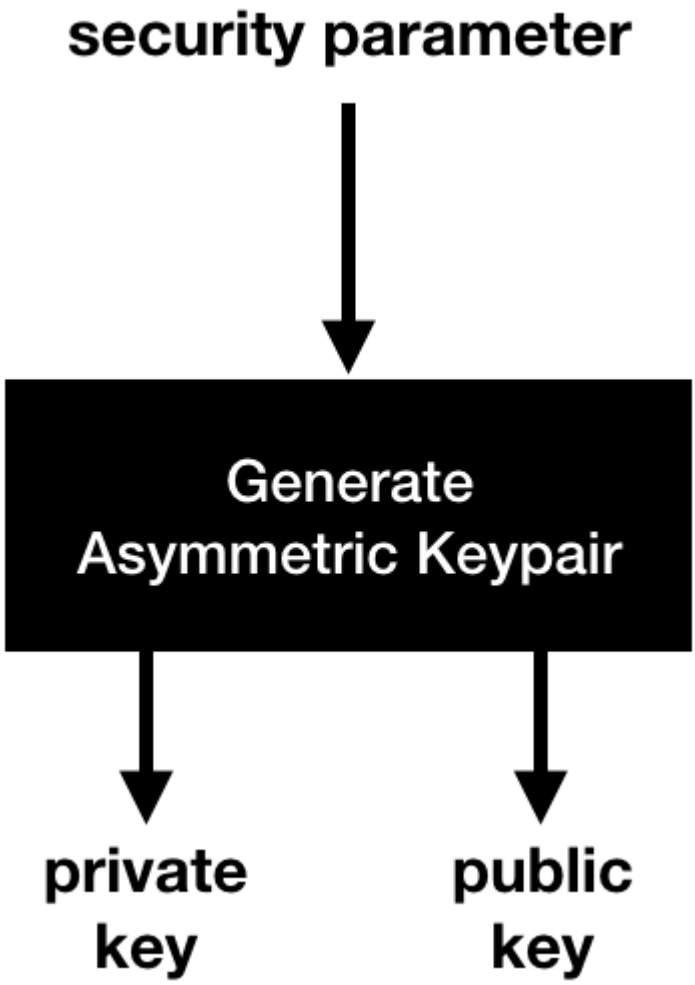


Figure 6.2 To use asymmetric encryption, you first need to generate a keypair. Depending on the security parameter you provide, you will generate keys of different security strength.

The key generation algorithm generates a keypair which comprises two different parts: the **public key** part (as the name indicates) can be published and shared without much concerns, while the **private key** must remain secret. Similar to the key generation algorithms of other cryptographic primitives, a security parameter is required in order to decide on the bit-security of the algorithm.

Anyone can then use the public key part to encrypt messages, and you can use the private key part to decrypt them as illustrated in figure 6.3.

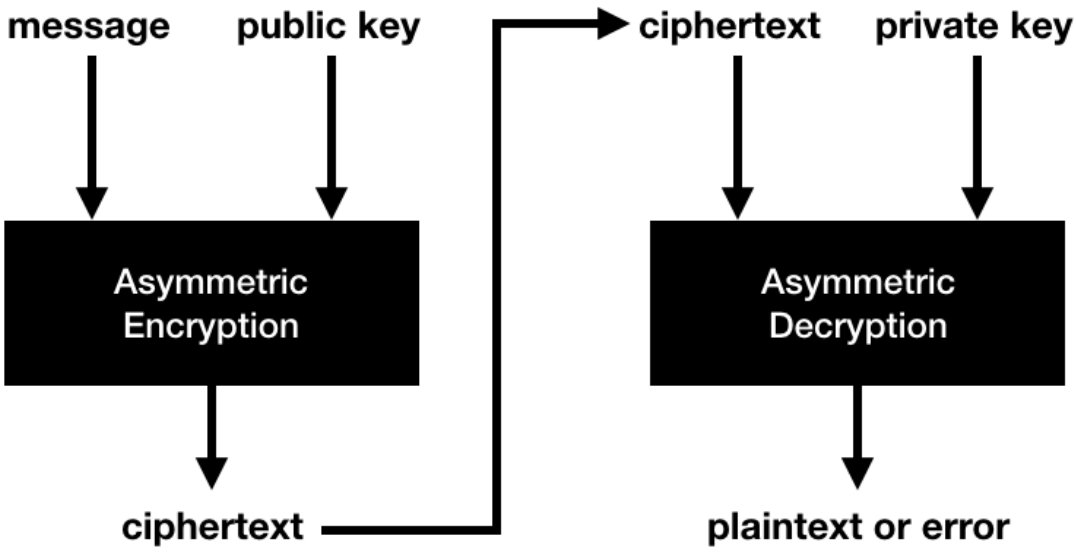


Figure 6.3 Asymmetric encryption allows one to encrypt a message (plaintext) using a recipient's public key. The recipient can then use a different algorithm to decrypt the encrypted message (ciphertext) using a private key related to the public key used previously.

Similar to authenticated decryption, decryption can fail if presented with an incoherent ciphertext.

> **WARNING** Note that no one is authenticated here by default: you know you are encrypting to a public key (which you think is owned by Alice) and Alice does not know for sure who sent this message.

Like in chapter 5, we still need to understand where the **trust** comes from. For now, we will imagine that we've obtained Alice's public key in real life, or from a friend, or from another secure mechanism we trust. In chapter 7, which covers digital signatures, I teach how real-world protocols solve this trust problem at scale.

Now assuming that **you know for sure that the public key you're using is Alice's**, the protocol is still not **mutually** authenticating the participants. Only **you** know you are encrypting to Alice, but Alice still has no way to verify your identity.

Let's now go to the next section, where you'll learn about how asymmetric encryption is used in practice (and spoiler alert, why it's rarely used in practice).

6.2 Asymmetric encryption in practice and hybrid encryption

You might be thinking: asymmetric encryption is probably enough to start encrypting your emails. In reality, asymmetric encryption is quite limited due to the limited length of messages it can encrypt. The speed of asymmetric encryption and decryption is also slow in comparison to symmetric encryption, this is due to asymmetric constructions implementing math operations, as opposed to symmetric primitives that often just manipulate bits. In this section you will learn about these limitations, what asymmetric encryption is actually used for in practice, and finally how cryptography overcomes these impediments. The section is divided in two parts for the two main use cases of asymmetric encryption:

- Key exchanges. You will see that it is quite natural to perform a key exchange (or key agreement) with an asymmetric encryption primitive.
- Hybrid encryption. The use cases for asymmetric encryption are quite limited due to the maximum size of what you can encrypt with it. To encrypt larger messages, you will learn about a more useful primitive called hybrid encryption.

6.2.1 Key Exchanges/Key Encapsulation

It turns out that asymmetric encryption can be used to perform a key exchange! The same kind as the ones we've seen in chapter 5. In order to do this, you can start by generating a symmetric key and encrypt it with Alice's public key—what we also call **encapsulating a key**—as seen in figure 6.4.

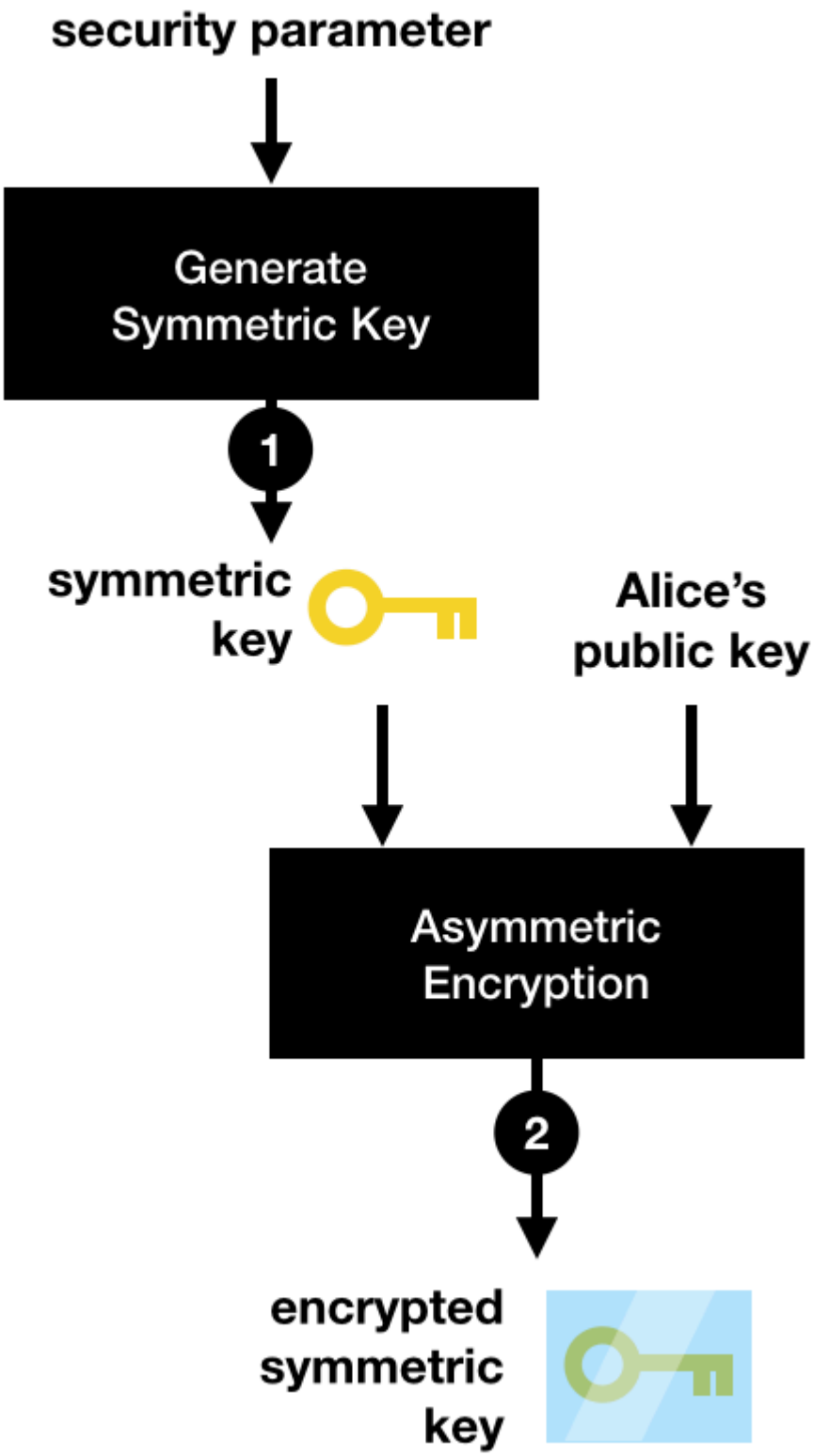


Figure 6.4 To use asymmetric encryption as a key exchange primitive, you first (1) generate a symmetric key and (2) encrypt it with Alice's public key.

You can then send the ciphertext to Alice who will be able to decrypt it and learn the symmetric

key. Subsequently, you will eventually both have a **shared secret**! The complete flow is illustrated in figure 6.5.

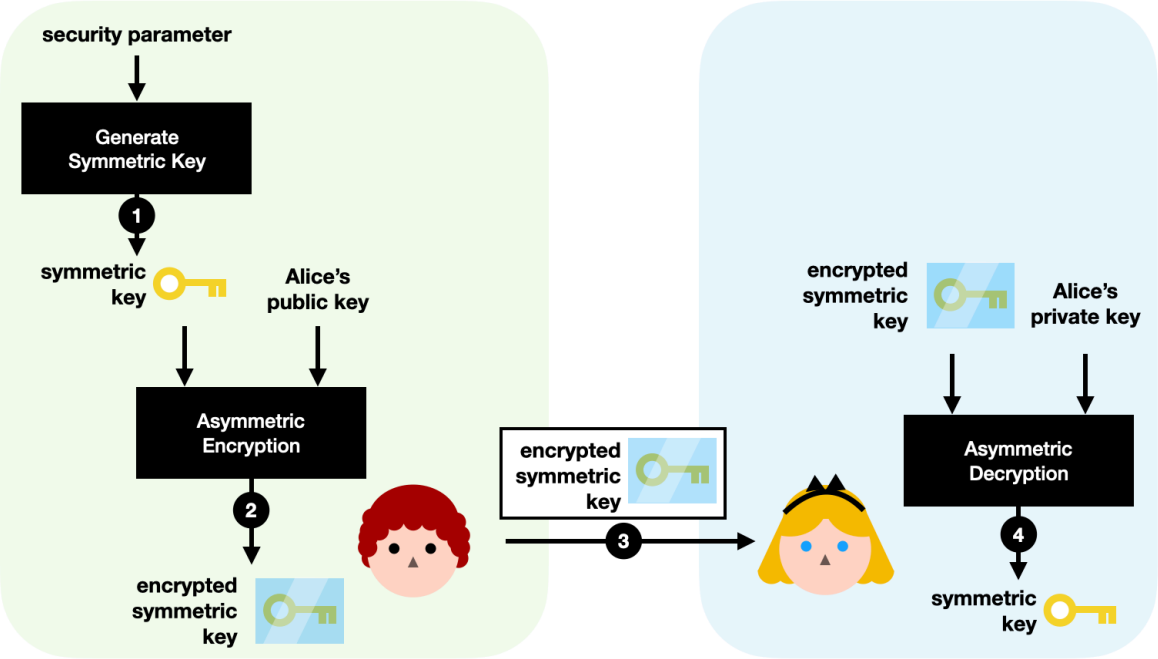


Figure 6.5 To use asymmetric encryption as a key exchange primitive, you can (1) generate a symmetric key and (2) encrypt it with Alice's public key. After (3) sending it to Alice, she can (4) decrypt it with her associated private key. At the end of the protocol they both have the shared secret, while no one else was able to derive it from observing the encrypted symmetric key alone.

Using asymmetric encryption to perform a key exchange is usually done with an algorithm called **RSA (following the names of its inventors Rivest, Shamir, and Adleman)**, and used in many internet protocols.

Today, RSA is often not the preferred way of doing a key exchange, and it is being used less and less in protocols in favor of Elliptic Curve Diffie-Hellman (ECDH). This is mostly due to historical reasons (many vulnerabilities have been discovered with RSA implementations and standards) and the attractiveness of the smaller parameter sizes offered by ECDH.

6.2.2 Hybrid Encryption

In practice, asymmetric encryption can only encrypt messages up to a certain length. For example, the size of plaintext messages that can be encrypted by RSA are limited by the security parameters that were used during the generation of the keypair (and more specifically by the size of the modulus). Nowadays, with the security parameters used (4096-bit modulus), the limit is approximately 500 ASCII characters. Pretty small. Therefore, most applications make use of hybrid encryption, whose limitation is tied to the encryption limits of the authenticated encryption algorithm used.

In practice, hybrid encryption has the same interface as asymmetric encryption (see figure 6.6).

People can encrypt messages with a public key and the one who owns the associated private key can decrypt the encrypted messages. The real difference is in the size limitations of the message that can be encrypted.

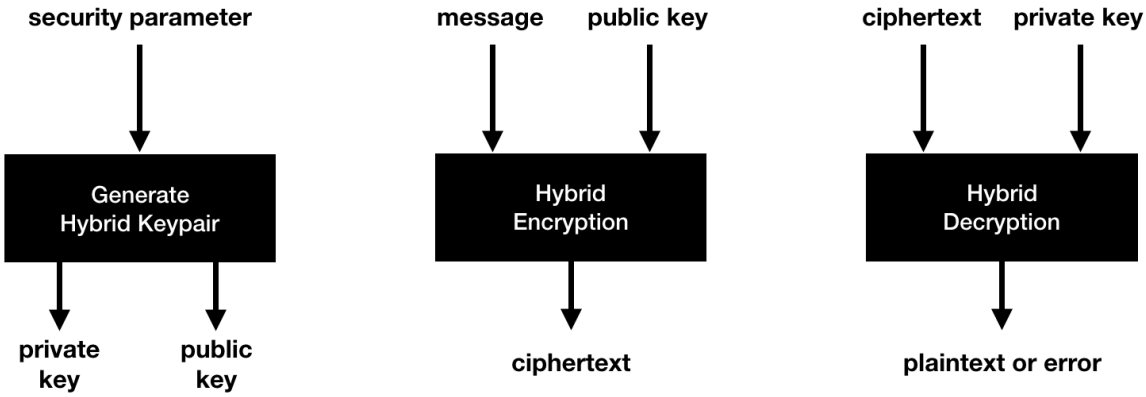


Figure 6.6 Hybrid Encryption has the same interface as asymmetric encryption, except that messages that can be encrypted are much larger in size.

Under the cover, hybrid encryption is simply the combination of an **asymmetric** cryptographic primitive with a **symmetric** cryptographic primitive (hence the name). Specifically, it is a non-interactive key exchange with the recipient followed by the encryption of a message with an authenticated encryption algorithm.

> **NOTE** Remember chapter 4 on authenticated encryption. You could have used a simple symmetric encryption primitive instead of an authenticated encryption primitive, but symmetric encryption does not protect against someone tampering your encrypted messages by default. This is why we never use symmetric encryption alone in practice.

Let's learn about how hybrid encryption works!

If you want to encrypt a message to Alice, you first generate a symmetric key and encrypt your message with it and an authenticated encryption algorithm as seen in figure 6.7.

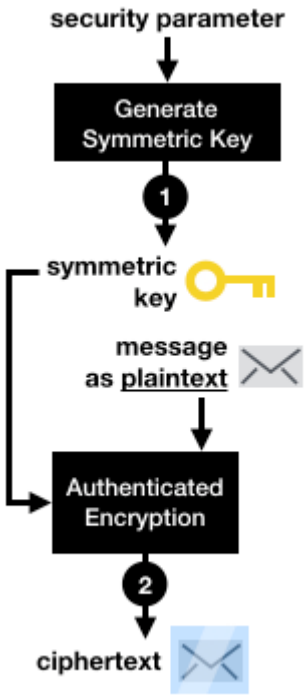


Figure 6.7 To encrypt a message to Alice using Hybrid Encryption with asymmetric encryption, you first (1) generate a symmetric key for an authenticated encryption algorithm. (2) You use the symmetric key to encrypt your message to Alice.

Once you have encrypted your message, Alice still cannot decrypt it without the knowledge of the symmetric key. How do we provide Alice with that symmetric key? Asymmetrically encrypt the symmetric key with Alice's public key as in figure 6.8.

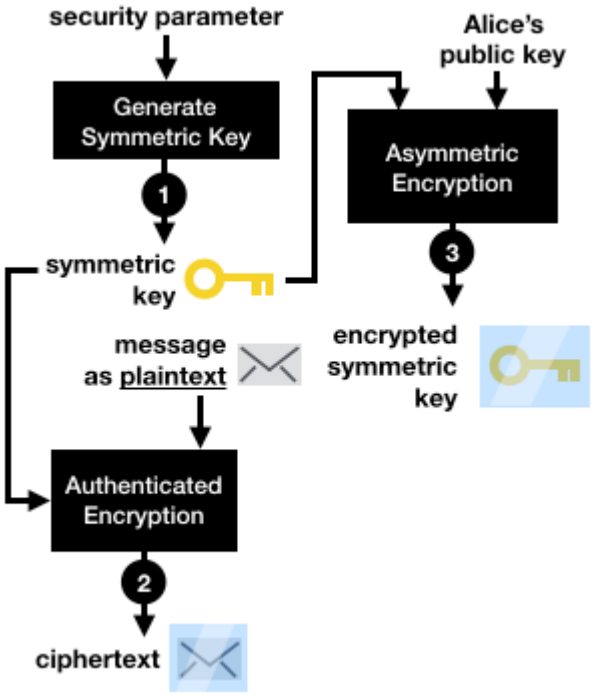


Figure 6.8 Building on figure 6.7. (3) You can encrypt the symmetric key itself by using Alice's public key and an asymmetric encryption algorithm.

Finally, you can send both of the results to Alice:

- the asymmetrically encrypted symmetric key.
- the symmetrically encrypted message.

which is enough information for Alice to decrypt the message. I illustrate the full flow in figure 6.9.

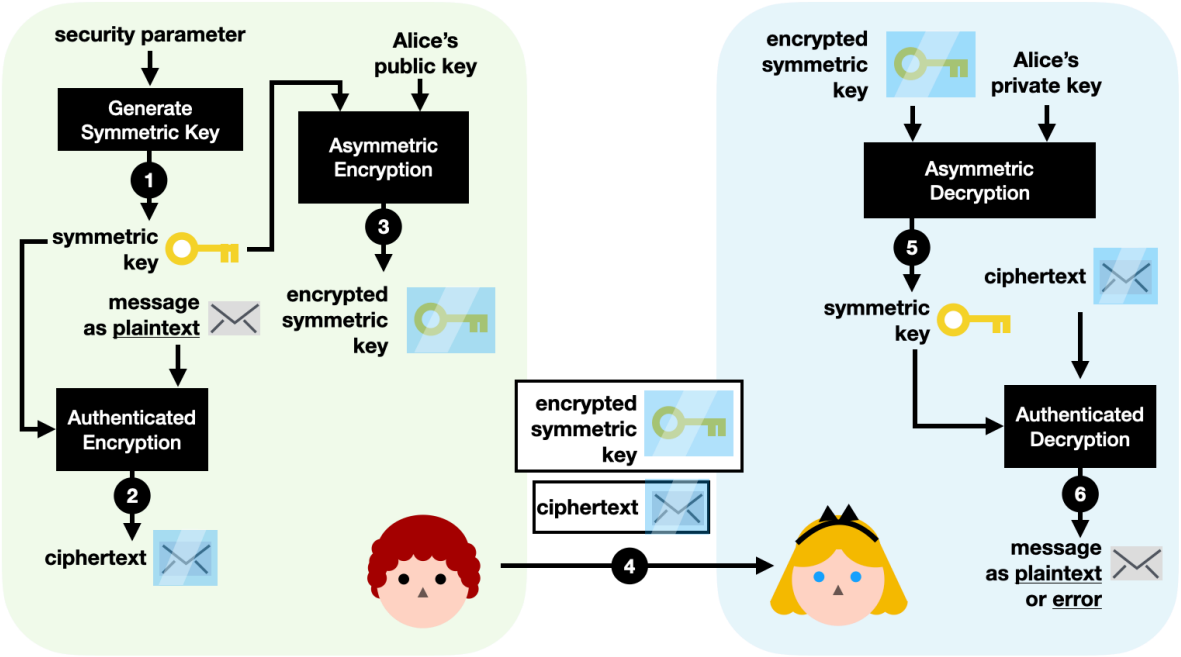


Figure 6.9 Building on figure 6.8. (4) After you send both the encrypted symmetric key and the encrypted message to Alice, (5) Alice can decrypt the symmetric key using her private key. (6) She can then use the symmetric key to decrypt the encrypted message. Step 5 and Step 6 can both fail and return errors if the communications were tampered with by a man-in-the-middle attacker at step 4.

And this is how we can use the best of both worlds, mixing asymmetric encryption and symmetric encryption to encrypt large amounts of data to a public key.

We often call the first asymmetric part of the algorithm a **Key Encapsulation Mechanism (KEM)** and the second symmetric part a **Data Encapsulation Mechanism (DEM)**.

Before we move to the next section and learn about the different algorithms and standards that exist for both asymmetric encryption and hybrid encryption, let's see in practice how you can use a cryptographic library to perform hybrid encryption.

To do this, I chose the **Tink** cryptographic library. Tink was developed by a team of cryptographers at Google to support large teams inside and outside of the company. Because of the scale of the project, conscious design choices were made and sane functions were exposed in order to prevent developers from mis-using cryptographic primitives. In addition, Tink is available in several programming languages (Java, C++, Obj-C, and Golang).

Listing 6.1 hybrid_encryption.java

```
import com.google.crypto.tink.HybridDecrypt;
import com.google.crypto.tink.HybridEncrypt;
import com.google.crypto.tink.hybrid.HybridKeyTemplates;
import com.google.crypto.tink.KeysetHandle;

KeysetHandle privateKeysetHandle = KeysetHandle.generateNew(
    HybridKeyTemplates.ECIES_P256_HKDF_HMAC_SHA256_AES128_GCM);   ❶

KeysetHandle publicKeysetHandle =
    privateKeysetHandle.getPublicKeysetHandle();   ❷

HybridEncrypt hybridEncrypt =
    publicKeysetHandle.getPrimitive(HybridEncrypt.class);   ❸
byte[] ciphertext = hybridEncrypt.encrypt(plaintext, associatedData);   ❸

HybridDecrypt hybridDecrypt = privateKeysetHandle.getPrimitive(
    HybridDecrypt.class);   ❹
byte[] plaintext = hybridDecrypt.decrypt(ciphertext, associatedData);   ❹
```

❶ We generate keys for a specific Hybrid Encryption scheme.

❷ We obtain the public key part that we can publish or share.

❸ Anyone who knows this public key can use it to encrypt a plaintext as well as authenticate some associated data.

❹ We can decrypt such an encrypted message using the same associated data. if the decryption fails, it will throw an exception that you have to catch. (This is specific to Java.)

One note to help you understand the `ECIES_P256_HKDF_HMAC_SHA256_AES128_GCM` string:

- **ECIES (for Elliptic Curve Integrated Encryption Scheme)** is the hybrid encryption standard to use, you'll learn about this later in this chapter. The rest of the string lists the algorithms used to instantiate ECIES.
- **P256** is the NIST standardized elliptic curve you've learned about in chapter 5 on key exchanges.
- **HKDF** is a key derivation function you will learn about in chapter 8 on randomness and secrets.
- **HMAC** is the message authentication code you've learned about in chapter 3.
- **SHA-256** is the hash function you've learned about in chapter 2.
- **AES-128-GCM** is the AES-GCM authenticated encryption algorithm using a 128-bit key you've learned about in chapter 4.

See how everything is starting to fit together?

And that's it. In the next section you will learn about RSA and ECIES, the two widely adopted standards for asymmetric encryption and hybrid encryption.

6.3 Asymmetric encryption with RSA, the bad and the less bad

It is time for us to look at the standards that define asymmetric encryption and hybrid encryption in practice. Historically, both of these primitives have not been spared by cryptanalysts, and many vulnerabilities and weakenesses have been found in both standards and implementations. This is why I will start this section with an introduction of the RSA asymmetric encryption algorithm, and how not to use it. The rest of the chapter will go through the actual standards you can follow to use asymmetric and hybrid encryption:

- **RSA-OAEP**. The main standard to perform asymmetric encryption with RSA.
- **ECIES.** The main standard to perform hybrid encryption with Elliptic Curve Diffie-Hellman.

6.3.1 Textbook RSA

In this section you will learn about the RSA public-key encryption algorithm, and how it has been standardized throughout the years. This will be useful in order to understand other secure schemes based on RSA.

Unfortunately, RSA has caught quite some bad rap since it was first published in 1977. One of the popular theories is that RSA is too easy to understand and implement, and thus many people do it themselves which leads to a lot of vulnerable implementations. But this is not the whole story! While the concept of RSA (often called "textbook RSA") is insecure if implemented naively, even some standards have been found to be insecure!

To understand these issues with RSA, you first need to learn how RSA works.

Remember the multiplicative group of numbers modulo a prime p (we've talked about it in chapter 5). It is the set of strictly positive integers:

1, 2, 3, 4, …, p-1

Let's imagine that **one of these numbers is our message**. For p large enough, let's say 4096-bit, our message can contain around 500 characters.

> **NOTE** **For computers, a message is just a series of bytes, which can also be interpreted as a number.**

We have seen that by exponentiating a number (let's say our message), we can **generate** other numbers which forms a **subgroup**. I illustrate this in figure 6.10.

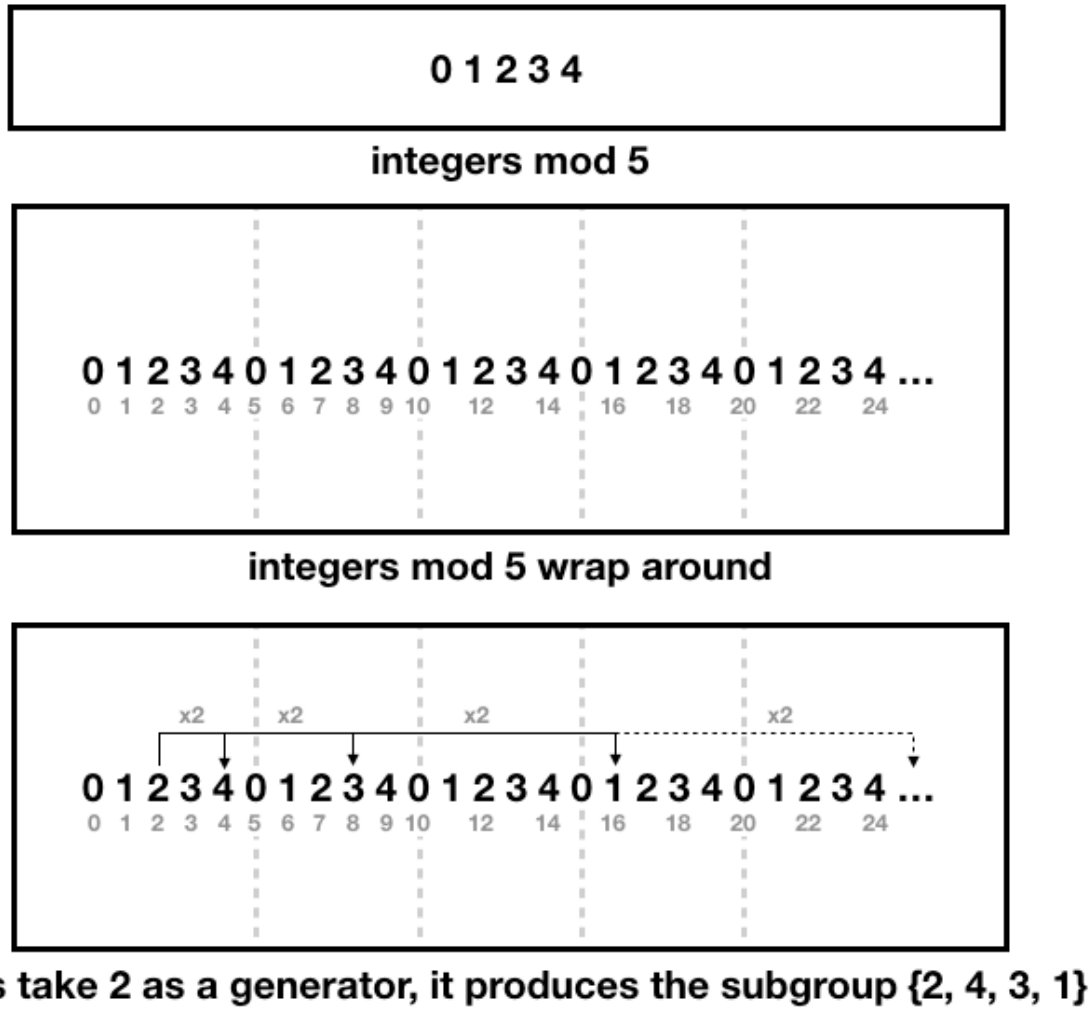


Figure 6.10 Integers modulo a prime (here 5) are divided in different subgroups. By picking an element as a generator (let's say the number 2) and exponentiating it, we can generate a subgroup. For RSA, the generator is the message.

This is useful for us to define how to encrypt with RSA. To do this, we publish a **public exponent** e (for *encryption*) and a prime number p. To encrypt a message m, one computes

ciphertext = m^e mod p

For example, to encrypt the message m=2 with e=2 and p=5, we do

ciphertext = 2^2 mod 5 = 4

And **this is the idea behind encryption with RSA!**

NOTE	Usually, a small number is chosen as public exponent e so that encryption is fast. Historically, standards and implementations seem to have settled on the prime number 65537 for the public exponent.

This is great, you now have a way for people to encrypt messages to you. But **how do you decrypt?**

Remember, if you continue to exponentiate a generator, you actually go back to the original number (see figure 6.11).

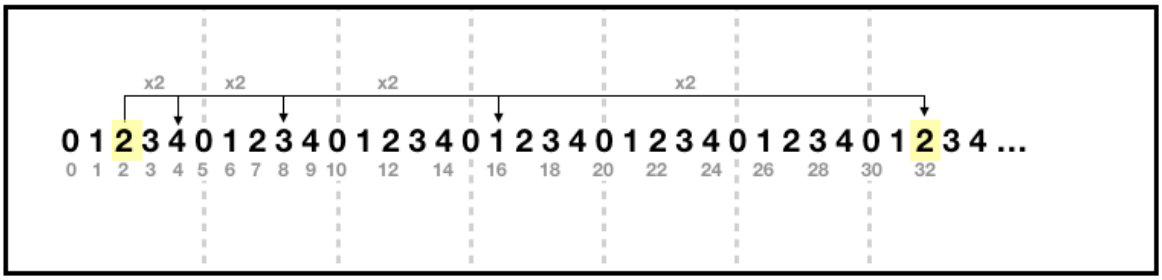


Figure 6.11 Let's say that our message is the number 2. By exponentiating it we can obtain other numbers in our group. If we exponentiate it enough, we go back to our original message 2. We say that the group is cyclic. This property can be used to recover a message after it has been raised to some power.

This should give you an idea of how to implement decryption: find out how much you need to exponentiate a ciphertext in order to recover the original generator (which is the message).

Let's say that you know such a number that we'll call the **private exponent d** (for *decryption*). If you receive ciphertext = message^e mod p, you should be able to raise it to the power d to recover the message:

$$\text{ciphertext}^d = (\text{message}^e)^d = \text{message}^{e \times d} = \text{message} \bmod p$$

The actual mathematics behind finding this private exponent d is a bit tricky, but simply put you compute the inverse of the public exponent modulo the order (number of elements) of the group:

$$d = e^{-1} \bmod \text{order}$$

We have an efficient algorithm to compute modular inverses,like the Extended Euclidean algorithm, and so this is not a problem.

We do have a problem though!

For a prime p, the order is simply p-1, and thus **the private exponent is easy to calculate for anyone**. This is because every element in this equation, besides d, is public.

NOTE How did we obtain the previous equation to compute the private exponent `d`? Euler's theorem states that for `m` co-prime with p (meaning that they have no common factors)

$m^{order} = 1 \bmod p$

For `order` the number of elements in the multiplicative group created by the integers modulo p. This implies in turn that for any integer `multiple`

$m^{1+multiple \times order} = m \times (m^{order})^{multiple} \bmod p = m \bmod p$

This tells us that the equation we are trying to solve

$m^{e \times d} = m \bmod p$

can be reduced to

$e \times d = 1 + multiple \times order$

which can be rewritten as

$e \times d = 1 \bmod order$

which by definition means that d is the inverse of e modulo order

One way we could prevent others from computing the private exponent from the public exponent is to **hide the order of your group**, and this is the brilliant idea behind RSA: if our modulus is not a prime anymore, but a **product of prime** $N=p \times q$ (with p and q large primes that are known only to you), then **the order of our multiplicative group is not easy to compute anymore as long as you don't know p and q**!

NOTE The order of the multiplicative group modulo a number N can be calculated with Euler's totient function (N) which returns the count of numbers that are coprime with N. (5 and 6 are coprime for example, because the only positive integer that divides both of them is 1. On the other hand, 10 and 15 are not, because 1 and 5 divides them both.)

The order of a multiplicative group modulo an RSA modulus $N=p \times q$ is $(N)=(p-1) \times (q-1)$ which is too hard to calculate unless you know the factorization of N.

We're all good! To recapitulate, this is how RSA works:

- **Key Generation**.
  - Generate two large prime numbers p and q.
  - Choose a random public exponent e, or a fixed one like e=65537.
  - Your public key is the public exponent e and the public modulus $N = p \times q$.
  - Derive your private exponent $d=e^{-1} \bmod (p-1)(q-1)$.
  - Your private key is the private exponent d.
- **Encryption**. To encrypt a message compute $message^e \bmod N$.
- **Decryption**. To decrypt a ciphertext compute $ciphertext^d \bmod N$.

Figure 6.12 recapitulates how RSA finally works in practice.

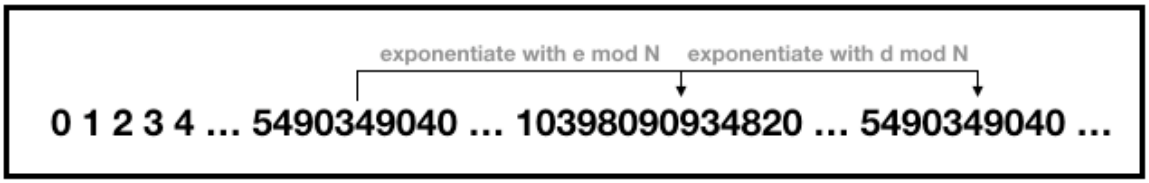


Figure 6.12 RSA encryption works by exponentiating a number (our message) with the public exponent e modulo the public modulus N=p×q. RSA decryption works by exponentiating the encrypted number with the private exponent d modulo the public modulus N.

We say that RSA relies on the **factorization problem**. Without the knowledge of p and q, no-one can compute the order, thus no-one but you can compute the private exponent from the public exponent. This is very similar to how Diffie-Hellman was based on the discrete logarithm problem, see figure 6.13.

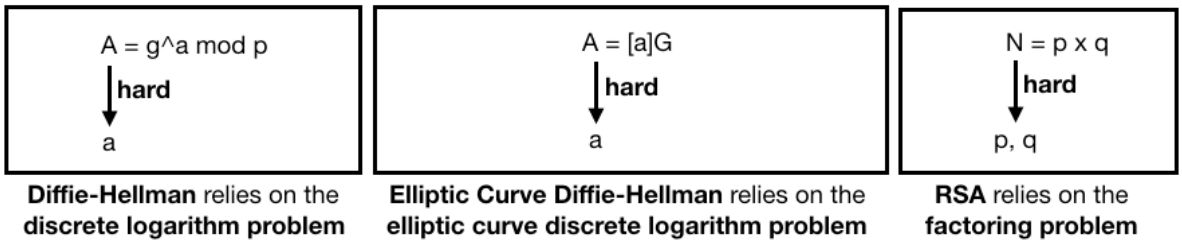


Figure 6.13 Diffie-Hellman, Elliptic Curve Diffie-Hellman and RSA are asymmetric algorithms that rely on three distinct problems in mathematics that we believe to be hard. "Hard" meaning that we do not know efficient algorithms to solve them when instantiated with large numbers.

Thus, textbook RSA works modulo a composite number N=p×q where p and q are two large primes that need to remain secret.

Now that you understand how RSA works, let's see how insecure it is in practice and what standards do to make it secure.

6.3.2 Why not to use RSA PKCS#1 v1.5

You've learned about "textbook RSA", which is insecure by default for many reasons. Before you can learn about the secure version of RSA, let's see what you need to avoid.

There are many reasons why you cannot use textbook RSA directly, one example is that if you encrypt small messages to Alice (for example m=2), then I can encrypt all the small numbers between 0 and 100 (for example) and observe very quickly that the encryption of the number 2 matches your ciphertext. Standards fix this issue by maximizing the size of your message with a **non-deterministic** padding before encrypting it. For example, the **RSA PKCS#1 v1.5** standard

defines a padding that adds a number of random bytes to the message before using RSA to encrypt it. I illustrated this in figure 6.14.

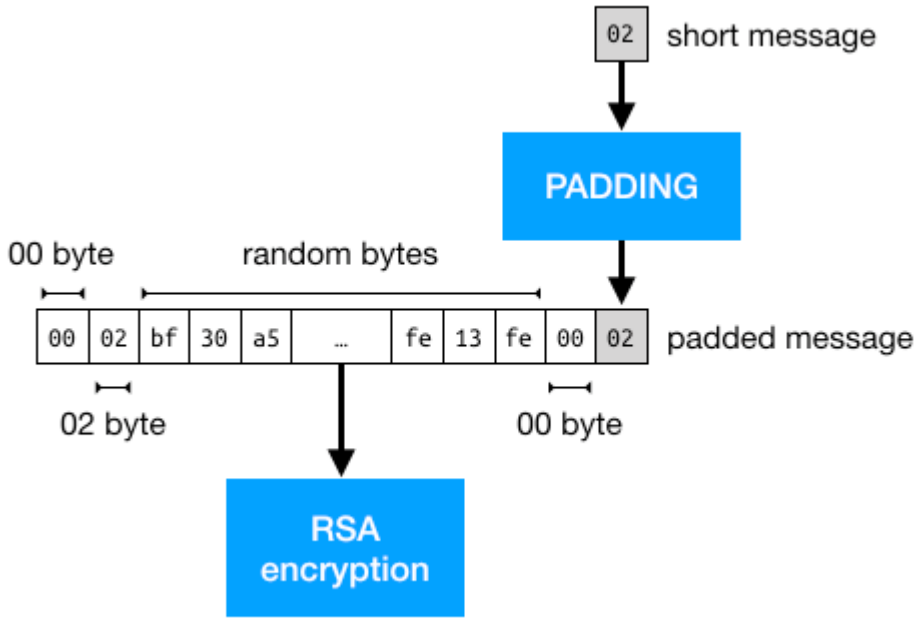


Figure 6.14 The RSA PKCS#1 v1.5 standard specifies a padding to apply to a message prior to encryption. The padding must be reversible (so that decryption can get rid of it) and must add enough random bytes to the message in order to avoid brute-force attacks.

The **PKCS#1** standard is actually the first standard on RSA, published as part of a series of **Public Key Cryptography Standard** written by the RSA company in the early 90's. While the PKCS#1 standard fixes some known issues, in 1998 **Bleichenbacher** found a practical attack on **PKCS#1 version 1.5** that allowed an attacker to decrypt messages encrypted with the padding specified by the standard. As it required a million messages it is infamously called the **Million Message Attack**. Mitigations were later found, but interestingly along the years the attack has been rediscovered again and again as researchers found that the mitigations were too hard to implement securely (if at all).

> **NOTE** Bleichenbacher's attack is a type of attack called an adaptive chosen-ciphertext attack (CCA2) in theoretical cryptography. CCA2 means that to perform this attack, an attacker is in a context that allows him or her to submit arbitrary RSA encrypted messages (chosen-ciphertext), observe how it influences the decryption, and continue the attack based on previous observation (adaptive). CCA2 is often used to model attackers in cryptographic security proofs.

To understand why the attack was possible, you need to understand that RSA ciphertexts are **malleable**: you can tamper with an RSA ciphertext without invalidating its decryption.

If I observe the ciphertext $c = m^e \bmod N$, then I can submit the following ciphertext:

$3^e \times m^e = (3m)^e \bmod N$

which will decrypt as:

$((3m)^e)^d = (3m)^{e \times d} = 3m \bmod N$

I used the number 3 as an example, but I can multiply the original message with whatever number I want. In practice, a message must be well-formed (due to the padding) and thus, tampering with a ciphertext should break the decryption. Nevertheless, it happens that sometimes, even after that transformation, the padding is accepted after decryption.

Bleichenbacher made use of this property in his Million Message Attack on RSA PKCS#1 v1.5. His attack works by intercepting an encrypted message, modifying it and sending it to the person in charge of decrypting it. By observing if that person can decrypt it (the padding remained valid), we obtain some information about the range in which the message is (since the first two bytes are `0x0002`, we know that the decryption is smaller than some value). By doing this iteratively, we can narrow that range down to the original message itself.

Even though the Bleichenbacher attack is well-known, there are still many systems in use today that implement RSA PKCS#1 v1.5 for encryption. As part of my work as a security consultant, I found many applications that were vulnerable to this attack, so be careful!

6.3.3 Asymmetric encryption with RSA-OAEP

In 1998, version 2.0 of the same PKCS#1 standard was released with a new padding scheme for RSA called **Optimal Asymmetric Encryption Padding (OAEP)**. Unlike its precedessor—PKCS#1 v1.5—OAEP is not vulnerable to Bleichenbacher's attack and is thus a strong standard to use for RSA encryption nowadays. Let's see how OAEP works and prevents the previously discussed attacks.

First, let's mention that like most cryptographic algorithms OAEP comes with a key generation algorithm that takes a security parameter (as illustrated in figure 6.15).

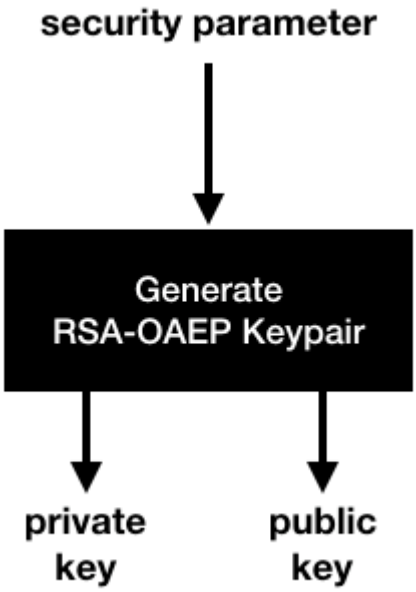


Figure 6.15 RSA-OAEP, like many public-key algorithms, first need to generate a keypair that can be used later in the other algorithms provided by the cryptographic primitive.

This algorithm takes a security parameter, which is a number of bits. As with Diffie-Hellman, operations happen in the set of numbers modulo a large number. When we talk about the security of an instantiation of RSA, we usually refer to the size of that large modulus. This is very similar to Diffie-Hellman if you remember. Currently, most guidances (see https://keylength.com) estimate a modulus between 2048 and 4096 bits to provide 128-bit security. As these estimations are quite different, most applications seem to conservatively settle on 4096-bit parameters.

> **NOTE** We've seen that RSA's large modulus is not a prime, but a product $N=p\times q$ of two large prime numbers p and q. For a 4096-bit modulus, the key generation algorithm typically splits things in the middle and generates both p and q of size approximately 2048-bit.

To encrypt, the algorithm first pads the message and mixes it with a random number generated per-encryption. The result is then encrypted with RSA. To decrypt the ciphertext, the process is reversed as can be seen in figure 6.16.

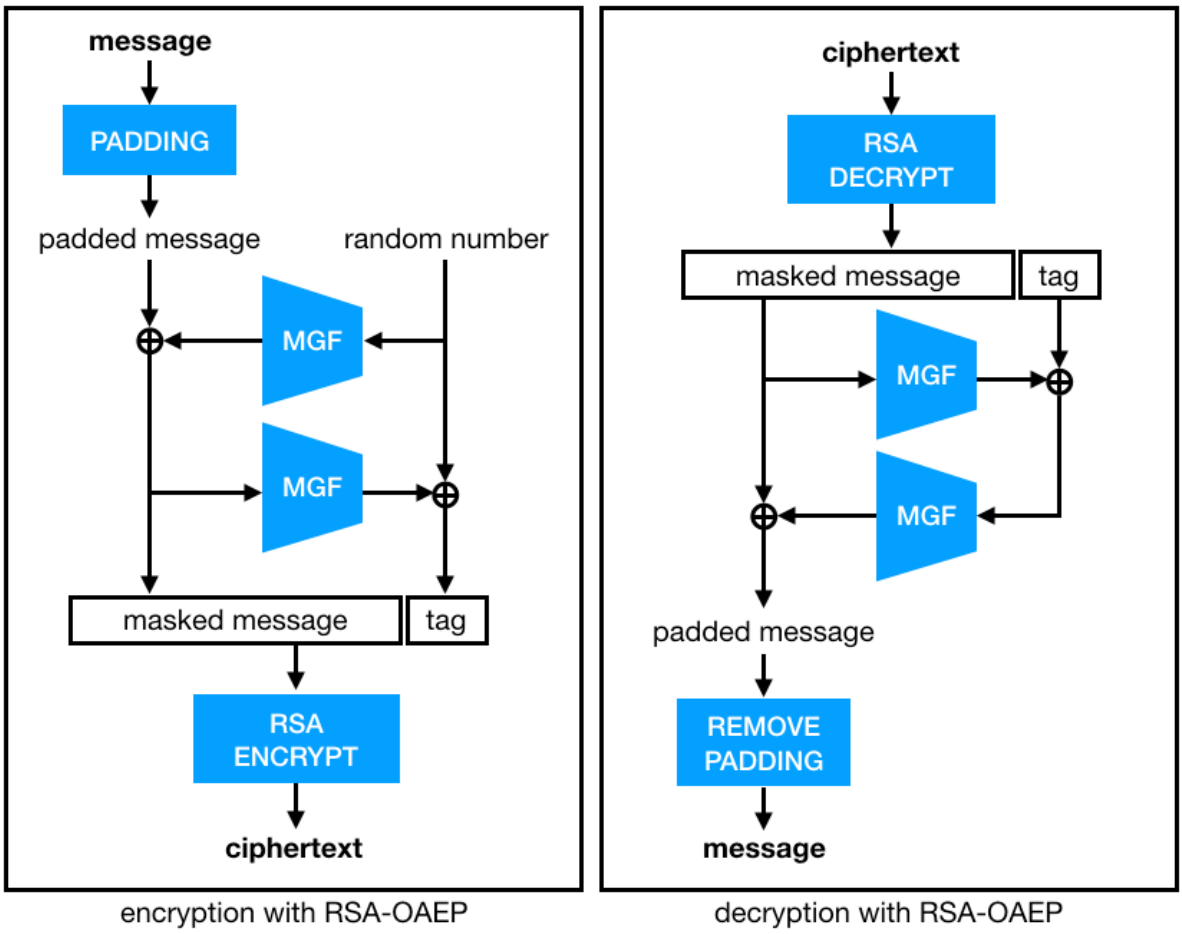


Figure 6.16 RSA-OAEP works by mixing the message with a random number prior to encryption. The mixing can be reverted after decryption. At the center of the algorithm, a Mask Generation Function (MGF) is used to randomize and enlarge or reduce an input.

RSA-OAEP uses this mixing in order to make sure that, if a few bits of what is encrypted with RSA leak, no information on the plaintext can be obtained. Indeed, to reverse the OAEP padding, you need to obtain (close to) all the bytes of the OAEP padded plaintext.

In addition, Bleichenbacher's attack should not work anymore as the scheme makes it impossible to obtain a well-formed plaintext by modifying a ciphertext.

> **NOTE** **The property that it is too difficult for an attacker to create a ciphertext that will successfully decrypt (of course without the help of encryption) is called plaintext-awareness. Due to the plaintext-awareness provided by OAEP, Bleichenbacher's attack do not work on the scheme.**

Inside of OAEP, **MGF** stands for **mask generation function**. In practice, a MGF is an **extendable output function (XOF)** (you've learned about them in chapter 2). As MGFs were invented before the sponge construction, they simply make use of a hash function that absorbs the input repeatedly with a counter (see figure 6.17).

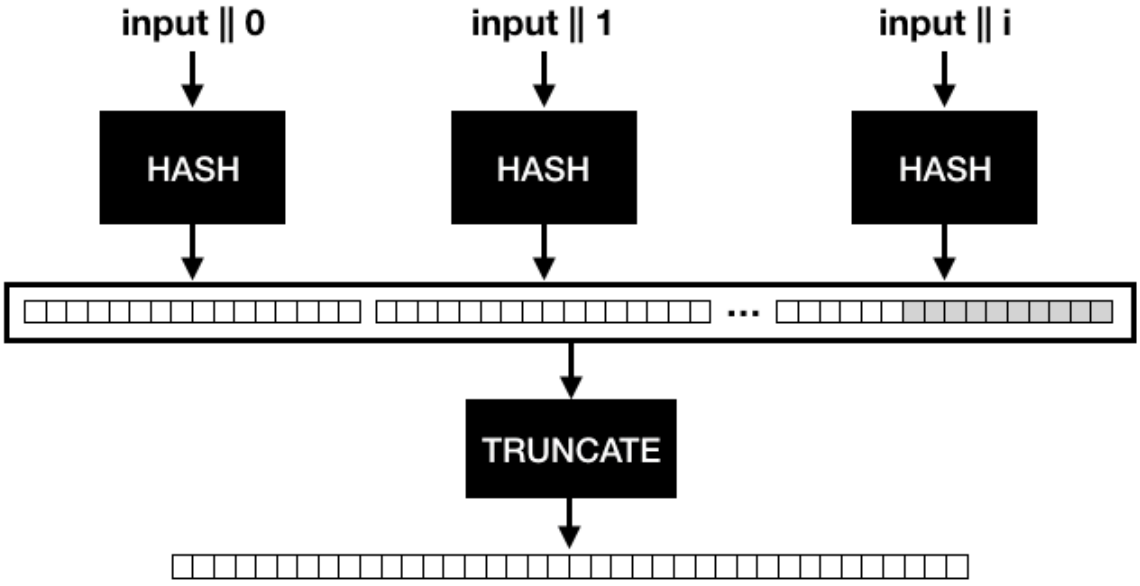


Figure 6.17 A mask generation function (MGF) is simply a function that takes an arbitrary-length input, and produces a random-looking arbitrary-length output. It works by hashing an input and a counter, concatenating the digests together, and truncating the result to obtain the length desired.

And this is how OAEP works.

> **NOTE** Only 3 years after the release of the OAEP standard, James Manger found a timing attack similiar to Bleichenbacher's Million Message attack (but much more practical) on OAEP if not implemented correctly. Fortunately, it is much simpler to securely implement OAEP compared to PKCS#1 v1.5 and vulnerabilities in this scheme's implementation are much more rare.

Furthermore the design of OAEP is not perfect, better constructions have been proposed and standardized over the years. One example is **RSA-KEM** which has stronger proofs of security and is much simpler to implement securely. You can observe how much more elegant the design is in figure 6.18.

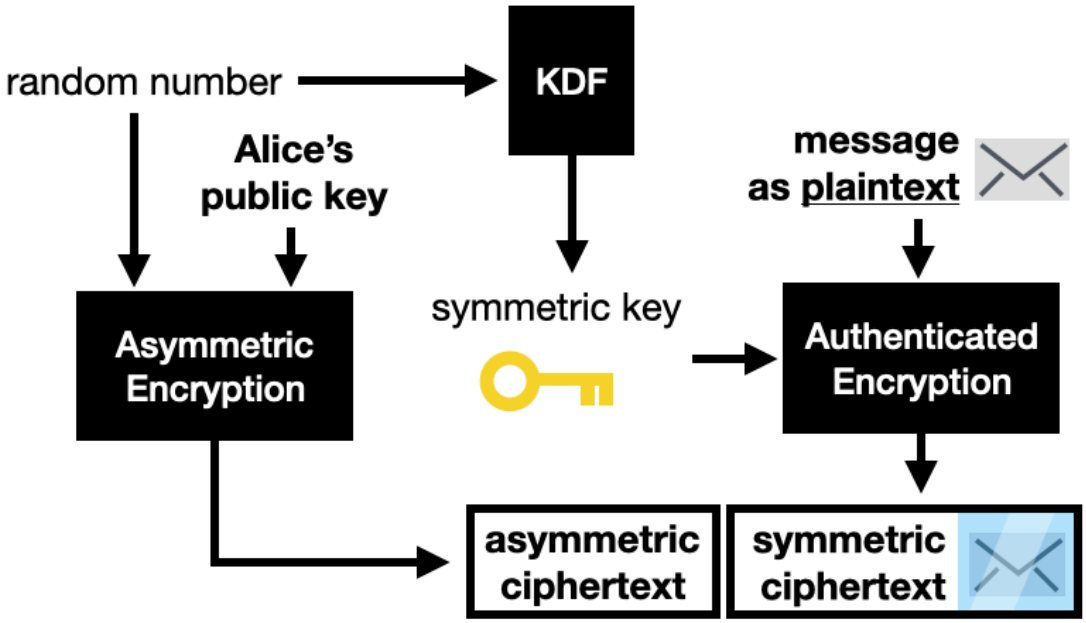


Figure 6.18 RSA-KEM is an encryption scheme that works by simply encrypting a random number under RSA. No padding is needed. The random number can be used as a symmetric key after being passed through a Key Derivation Function (KDF). The symmetric key is then used to encrypt a message via an authenticated encryption algorithm.

Note the **Key Derivation Function (KDF)** in use here. It is another cryptographic primitive that can be replaced with a MGF or a XOF. I'll talk more about what KDFs are in chapter 8 on randomness and secrets.

Nowadays most protocols and applications that use RSA either still implement the insecure PKCS#1 v1.5 or OAEP. On the other hand, more and more protocols are moving away from RSA encryption in favor of Elliptic Curve Diffie-Hellman (ECDH) for both key exchanges and hybrid encryption. This is understandable as ECDH provides shorter public keys and benefits in general from much better standards and much safer implementations.

6.4 Hybrid encryption with ECIES

While there exist many hybrid encryption schemes, the most widely adopted standard is **Elliptic Curve Integrated Encryption Scheme (ECIES)**. The scheme has been specified to be used with Elliptic Curve Diffie-Hellman, and has been included in many standards like ANSI X9.63, ISO/IEC 18033-2, IEEE 1363a, and SECG SEC 1. Unfortunately, every standard seems to implement a different variant, and different cryptographic libraries will implement hybrid encryption differently in part due to this.

For this reason, I've rarely seen two similar implementations of hybrid encryption in the wild. It is important to understand that while this is annoying, if all the participants of the protocol use the same implementation or document the details of the hybrid encryption scheme they have

implemented, then there are no issues.

ECIES works very similarly to the hybrid encryption scheme I've explained in section 6.2. The difference is that we implement the Key Encapsulation Mechanism (KEM) part with an ECDH key exchange instead of with an asymmetric encryption primitive.

Let's explain this step by step. First, if you want to encrypt a message to Alice, you use an (EC)DH-based key exchange with Alice's public key and a keypair that you generate for the occasion (this is called an **ephemeral keypair**). You can then use the obtained shared secret with an authenticated symmetric encryption algorithm like AES-GCM to encrypt a longer message to her. This is illustrated in figure 6.19.

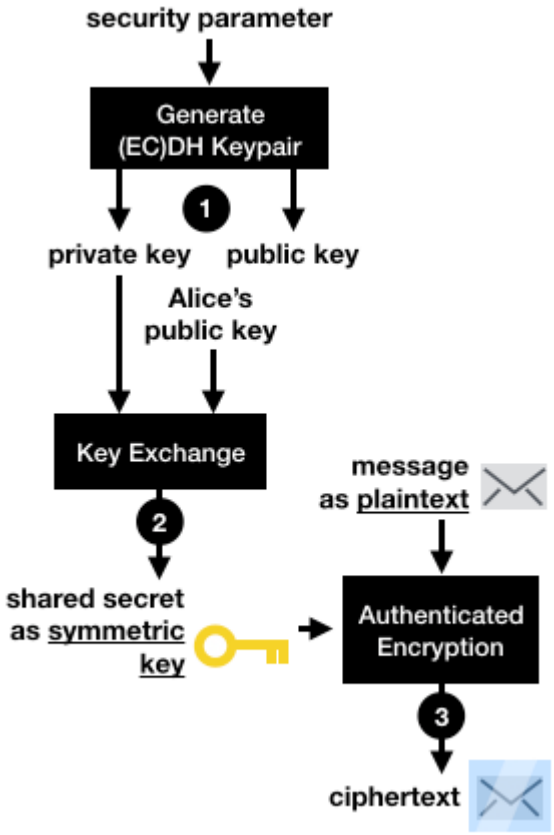


Figure 6.19 To encrypt a message to Alice using Hybrid Encryption with (EC)DH, you first (1) generate an ephemeral (Elliptic Curve) Diffie-Hellman key pair. (2) Perform a key exchange with your ephemeral private key and Alice's public key. (3) Use the resulting shared secret as a symmetric key to an authenticated encryption algorithm to encrypt your message.

After this, you can send the ephemeral public key and the ciphertext to Alice. Alice can use your ephemeral public key to perform a key exchange with her own keypair. She can then use the result to decrypt the ciphertext and retrieve the original message. The result is either the original message, or an error if the public key or the encrypted message were tampered in transit. The full flow is illustrated in figure 6.20.

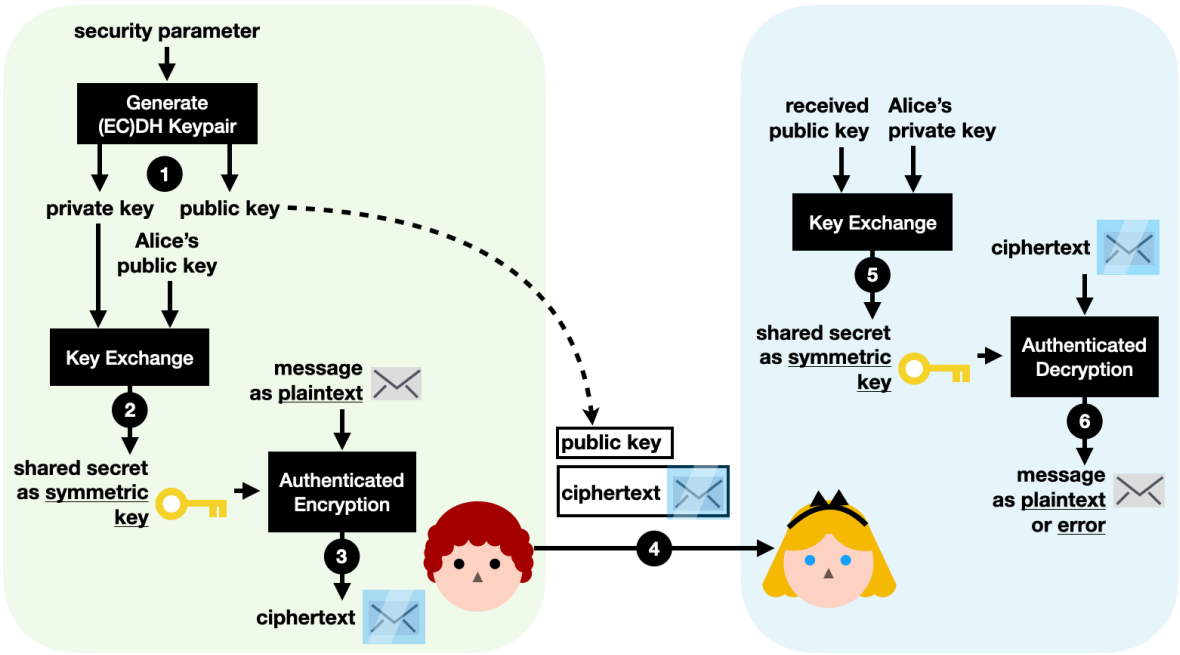


Figure 6.20 Building on figure 6.19, After (4) you send your ephemeral public key and your encrypted message to Alice. (5) Alice can perform a key exchange with her private key and your ephemeral public key. (6) She finally use the resulting shared secret as a symmetric key to decrypt the encrypted message with the same authenticated encryption algorithm.

And this is pretty much how ECIES work. There also exists a variant of ECIES using Diffie-Hellman called IES that works pretty much the same way, but not many people seem to use it.

> **NOTE** Note that I simplified the diagram above. Most authenticated encryption primitives expect an uniformly-random symmetric key. Since the output of a key exchange is generally not uniformly-random, we need to pass the shared secret through a Key Derivation Function (KDF) or a XOF beforehand. You will learn more about this in chapter 8. Not uniformly random here means that the probability that each bit of the key exchange result is 0 or 1 is not balanced. The first bits might always be set to 0, for example. This is an inherent consequence of using mathematics! Can you see why?

And that's it for the different standards you can use.

6.5 Summary

- Asymmetric encryption is rarely used to encrypt messages directly. This is due to the relatively low size limitations of that data that asymmetric encryption can encrypt.
- Hybrid Encryption can encrypt much larger messages by combining asymmetric encryption, or a key exchange, with a symmetric authenticated encryption algorithm.
- The RSA PKCS#1 v1.5 standard for asymmetric encryption is broken in most settings. Prefer the RSA-OAEP algorithm standardized in RSA PKCS#1 v2.2.
- ECIES is the most widely used hybrid encryption scheme. It is preferred over RSA-based schemes due to its parameter sizes and its reliance on solid standards.
- Different cryptographic libraries might implement hybrid encryption differently. This is not a problem in practice if interoperable applications use the same implementations.

7

Signatures and zero-knowledge proofs

This chapter covers

- Zero-knowledge proofs and how they can be used to simulate cryptographic signatures.
- The existing standards for signatures and how they are implemented in real-world applications.
- The subtle behaviors of signatures and how you can avoid their pitfalls.

You're about to learn one of the most ubiquitous and powerful cryptographic primitives: digital signatures. To put it simply: digital signatures are similar to the real-life signatures that you're used to, the ones that you scribe on checks and contracts. Except, of course, that they're cryptographic and so they provide much more assurance than their pen-and-paper equivalents.

In the world of protocols, digital signatures unlock so many different possibilities that you'll run into them again and again in the second part of this book. In this chapter I will introduce what this new primitive is, how it can be used in the real-world, and what the modern digital signature standards are. Finally, I will talk about security considerations and the hazards of using digital signatures.

NOTE **Signatures in cryptography are often referred to as digital signatures, or signature schemes. In this book, I interchangeably use these terms.**

For this chapter you'll need to have read:

- Chapter 2 on hash functions.
- Chapter 5 on key exchanges.
- Chapter 6 on asymmetric encryption.

7.1 What is a signature?

I explained in chapter 1 that cryptographic signatures are pretty much like real-life signatures. For this reason, they are usually one of the most intuitive cryptographic primitives to understand:

- Only you can use your signature to sign arbitrary messages.
- Anybody can verify your signature on a message.

As we're in the realm of asymmetric cryptography, you can probably guess how this asymmetry is going to take place. A signature scheme typically consists of three different algorithms:

1. A **keypair generation** algorithm that a signer uses to create a new private and public key (the public key can then be shared with anyone).
2. A **signing** algorithm that takes a private key and a message to produce a signature.
3. A **verifying** algorithm that takes a public key, a message, and a signature and returns a success or error message.

Sometimes the private key is also called the **signing key**, and the public key is called the **verifying key**. Makes sense right? I recapitulate these three algorithms in figure 7.1.

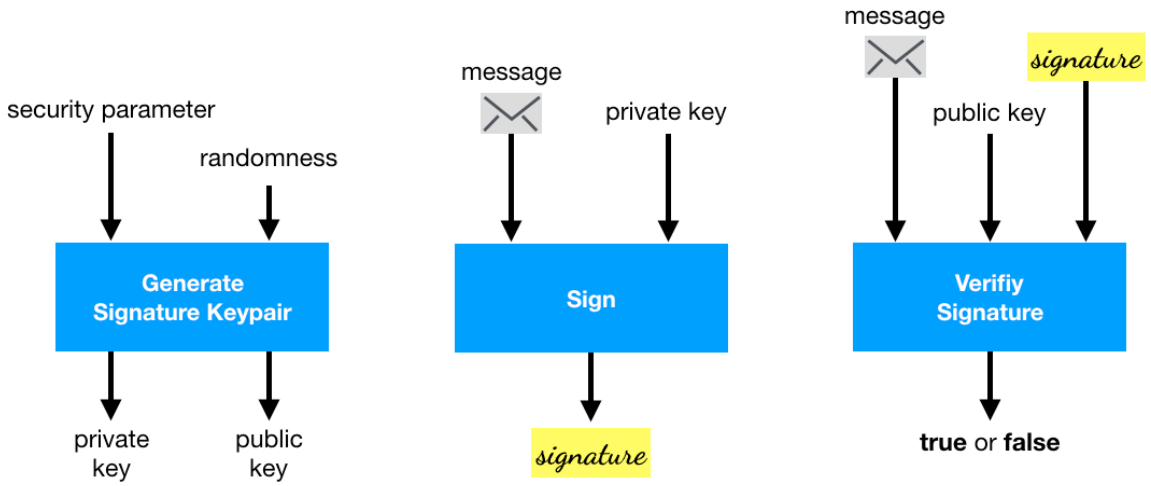


Figure 7.1 The interface of a digital signature. Like other public-key cryptographic algorithms, you first need to generate a keypair via a key generation algorithm that takes a security parameter and some randomness. You can then use a signing algorithm with the private key to sign a message, and a verifying algorithm with the public key to verify a signature over a message. You can't forge a signature that will verify under a public key, if you don't have access to its associated private key.

What are signatures good for? They are good for authenticating the origin of a message as well as the integrity of a message:

- **Origin**: if my signature is on it, it came from me.
- **Integrity**: if someone modifies the message, it'll void the signature.

> **NOTE** While these two properties are linked to authentication, they are often distinguished as two separate properties: origin authentication and message authentication (or integrity).

In a sense, signatures are similar to the message authentication codes (MACs) which you've learned about in chapter 3. But unlike MACs, they allow one to **authenticate messages asymmetrically**: a participant can verify that a message hasn't been tampered, without knowledge of the private or signing key.

> **NOTE** As you've seen in chapter 3, authentication tags produced by MACs must be verified in constant-time to avoid timing attacks. Do you think we need to do the same for verifying signatures?

Next, I'll show you how these algorithms can be used in practice.

7.1.1 How to sign and verify signatures in practice

Let's see a practical example. I chose to use *pyca/cryptography* (https://cryptography.io), a well-respected Python library. The following example simply generates a keypair, signs a message using the private key part, then verifies the signature using the public key part.

Listing 7.1 signature.py contains code to sign and verify signatures

```
from cryptography.hazmat.primitives.asymmetric.ed25519
import Ed25519PrivateKey   ❶

private_key = Ed25519PrivateKey.generate()   ❷
public_key = private_key.public_key()   ❷

message = b"example.com has the public key 0xab70..."   ❸
signature = private_key.sign(message)   ❸

try:   ❹
    public_key.verify(signature, message)   ❹
    print("valid signature")   ❹
except InvalidSignature:   ❹
    print("invalid signature")   ❹
```

❶ We use the Ed25519 signing algorithm, a popular signature scheme.

❷ With pyca/cryptography you first generate the private key, then generate the public key.

❸ Using the private key we can sign a message and obtain a signature.

❹ We verify the signature over the message using the public key part.

As I said earlier, digital signatures unlock many use cases in the real world. Let's see an example in the next section!

7.1.2 A prime use case for signatures: authenticated key exchanges

Chapters 5 and 6 have introduced different ways to perform key exchanges between two participants. You've learned in the same chapters that these key exchanges are useful to negotiate a shared secret, which can then be used to secure communications with an authenticated encryption algorithm. Yet, key exchanges didn't fully solve the problem of setting up a secure connection between two participants, as an active man-in-the-middle attacker could trivially impersonate both sides of a key exchange. This is where signatures enter the ring.

Imagine that Alice and Bob are trying to set up a secure communication channel between themselves, and that Bob is aware of Alice's verifying key. Knowing this, Alice can use her signing key to authenticate her side of the key exchange: she generates a key exchange keypair, signs the public key part with her signing key, then send the key exchange public key along with the signature. Bob can verify that the signature is valid using the associated verifying key he already knew, and then use the key exchange public key to perform a key exchange. We call such a key exchange an **authenticated key exchange**. If the signature is invalid, Bob can tell someone is actively man-in-the-middleing the key exchange. I illustrate authenticated key exchanges in figure 7.2.

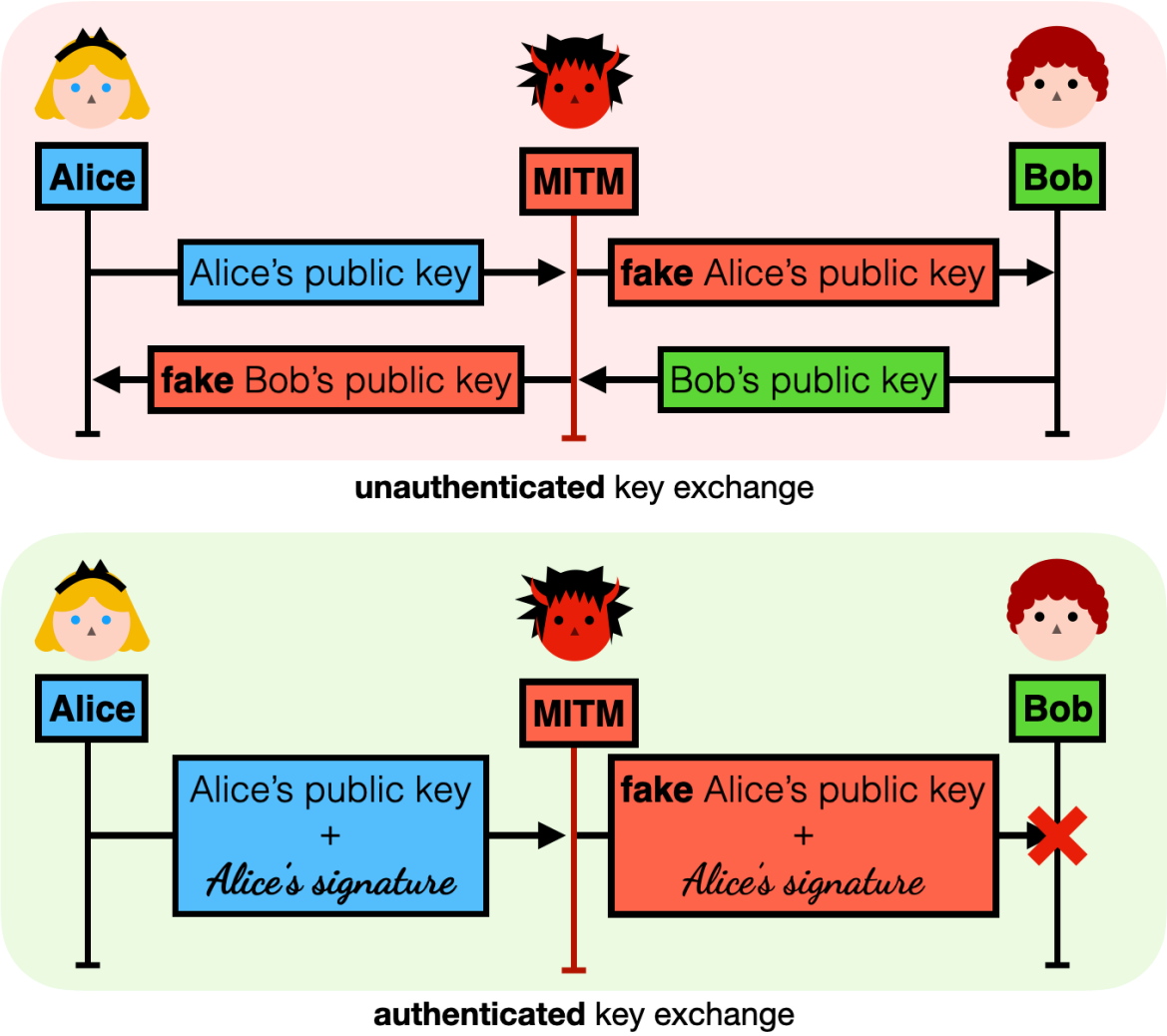


Figure 7.2 The first picture represents an unauthenticated key exchange, which is insecure to an active man-in-the-middle attacker who can trivially impersonate both sides of the exchange by swapping their public keys with their own. The second picture represents the beginning of a key exchange authenticated by Alice's signature over her public key. As Bob (who knows Alice's verifying key) is unable to verify the signature after the message was tampered by the man-in-the-middle attacker, he aborts the key exchange.

Note that in this example, the key exchange is only authenticated on one side: while Alice cannot be impersonated, Bob can. If both sides are authenticated (Bob would sign his part of the key exchange), we call the key exchange a **mutually-authenticated key exchange**.

Signing key exchanges might not appear super useful just yet: it seems like we moved the problem of not knowing Alice's key exchange public key in advance, to the problem of not knowing her verifying key in advance. The next section will introduce a real-world use of authenticated key exchanges that will make much more sense.

7.1.3 A real-world usage: public key infrastructures

Signatures become much more powerful if you assume that trust is **transitive**. By that, I mean that if you trust me, and I trust Alice, then you can trust Alice. She's cool.

Transitivity of trust allows you to scale trust in systems in extreme ways: imagine that you have confidence in some authority and their verifying key. Furthermore, imagine that this authority has signed messages indicating what the public key of Charles is, what the public key of David is, and so on. Then, you can choose to have faith in this mapping! Such a mapping is called a **public key infrastructure**.

For example, if you attempt to do a key exchange with Charles, and they claim that their public key is a large number that looks like 3848…; you can verify that by checking if your beloved authority has signed some message that looks like "The public key of Charles is: 3848…"

One real-world application of this concept is the **web public key infrastructure (web PKI)**. The web PKI is what your web browser uses to authenticate key exchanges it performs with the multitude of websites you visit every day.

A simplified explanation of the web PKI (illustrated in figure 7.3) is the following: when you download a browser, it comes with some verifying key baked into the program. This verifying key is linked to an authority whose responsibility is to sign thousands and thousands of websites' public keys, so that you can trust them without knowing about them. What you're not seeing is that these websites have to prove to the authority that they truly own their domain name, before they can get their public keys signed. (In reality, your browser trusts many authorities to do this job.)

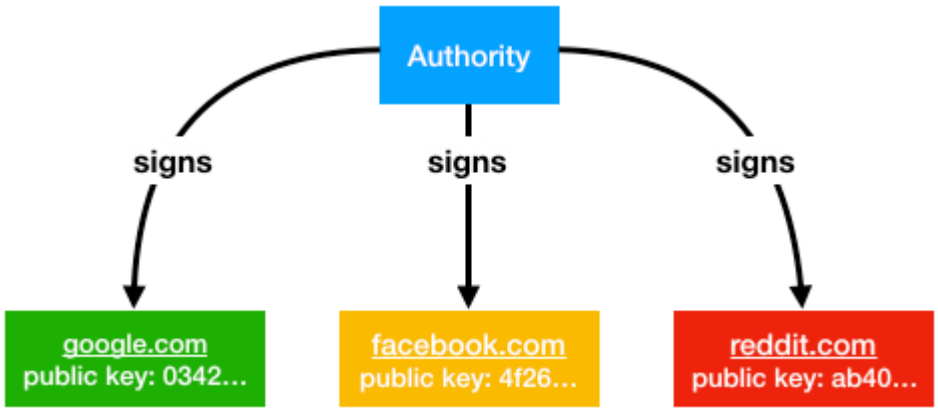


Figure 7.3 In the web PKI, browsers trust an authority to certify that some domains are linked to some public keys. When visiting a website securely, your browser can verify that the website's public key is indeed theirs (and not from some man in the middle), by verifying a signature from the authority.

In this section you have learned about signatures from a high-level point of view. Let's dig deeper into how signatures really work, but for this we first need to make a detour and take a

look at something called a zero-knowledge proof…

7.2 Zero-knowledge proofs: the origin of signatures

The best way to understand how signatures work in cryptography, is to understand where they come from. For this reason, let's take a moment to briefly introduce **zero-knowledge proofs (ZKPs)** and then I'll get back to signatures.

Imagine that Peggy wants to prove something to Victor. For example, she wants to prove that she knows the discrete logarithm to the base of some group element. In other words, she wants to prove that she knows x given $Y = g^x$ with g the generator of some group.

Of course, the simplest solution is for Peggy to simply send the value x (called the **witness**). This solution would be a simple **proof** of knowledge, and this would be OK unless Peggy does not want Victor to learn it.

> **NOTE** In theoretical terms, we say that the protocol to produce a proof is complete if Peggy can use it to prove to Victor that she knows the witness. If she can't use it to prove what she knows, then the scheme is useless, right?

In cryptography, we're mostly interested in proofs of knowledge that don't divulge the witness to the verifier. Such proofs are called **zero-knowledge proofs (ZKPs)**.

7.2.1 Schnorr identification protocol: an interactive zero-knowledge proof

In the next pages, I will build a zero-knowledge proof incrementaly from broken protocols, to show you how Alice can prove that she knows x without revealing x.

The typical way to approach this kind of problem in cryptography is to "hide" the value with some randomness, for example, by encrypting it. But we're doing more than just hiding: we also want to prove that it is there. So we need an algebraic way to hide it.

A simple solution is to simply add a randomly-generated value k to the witness:

$s = k + x$

Peggy can then send the hidden witness s, along with the random value k, to Victor.

At this point Victor has no reason to trust that Peggy did hide the witness in s. Indeed, if she doesn't know the witness x then s is probably just somee random value. What Victor does know, is that the witness x is hiding in the exponent of g, since he knows $Y = g^x$.

To see if Peggy dose know the witness, Victor can check if what she gave him matches what he knows, and this has to be done in the exponent of g as well (since this is where the witness is). In other words, Victor checks that these two numbers are equal:

- g^s $(= g^{k+x})$
- $Y \times g^k$ $(= g^x \times g^k = g^{x+k})$

The idea is that only someone who knows the witness x could have constructed a blinded witness s that satisfies this equation. And as such, it's a proof of knowledge. I recapitulate this zero-knowledge prooving system in figure 7.4.

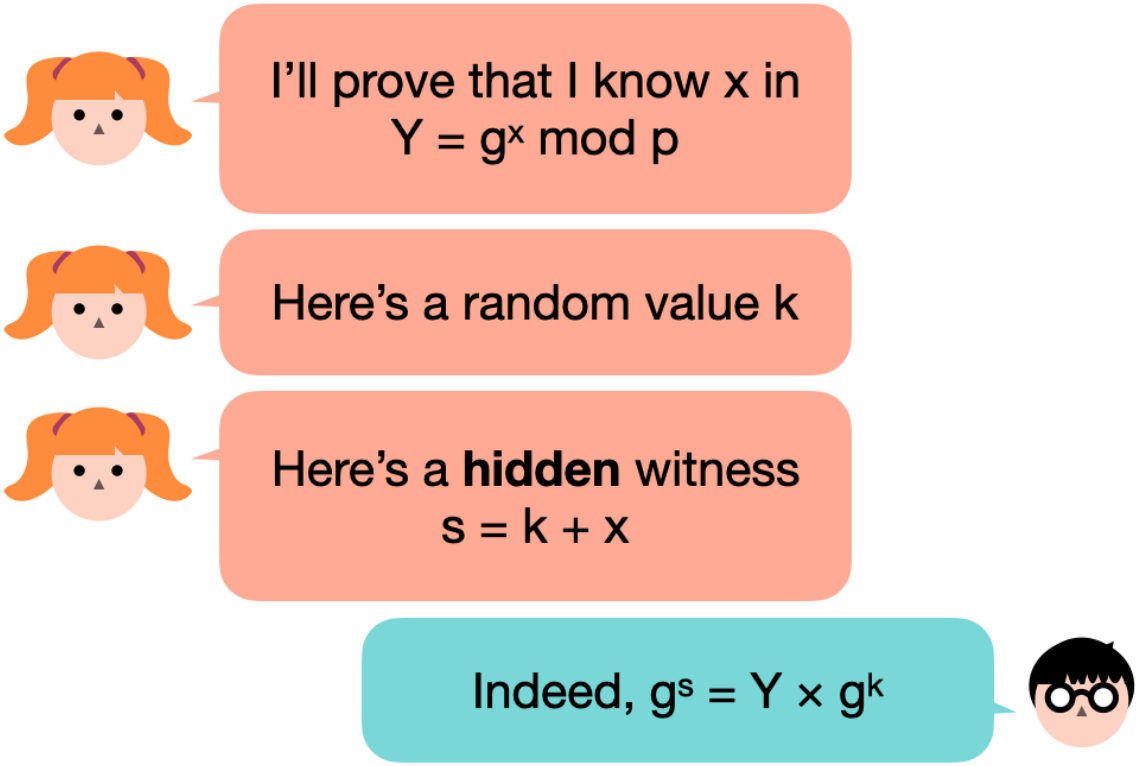


Figure 7.4 In order to prove to Victor that she knows a witness x, Peggy hides it (by adding it to a random value k) and sends the hidden witness s instead.

Not so fast… There's one problem with this scheme: it's obviously not secure. Indeed, since the

equation hiding the witness x only has one unknown (x itself), Victor can simply reverse the equation to retrieve the witness:

$x = s - k$

To fix this, Peggy can hide the random value k itself! This time, she has to hide the random value in the exponent (instead of addinig it to another random value) to make sure that Victor's equation still work:

$R = g^k$

This way, Victor does not learn the value k (this is the discrete logarithm problem, covered in chapter 5), and thus cannot recover the witness x. Yet, he still has enough information to verify that Peggy knows x! Victor simply has to check that g^s $(= g^{k+x} = g^k \times g^x)$ is equal to $Y \times R$ $(= g^x \times g^k)$. I recapitulate this second attempt at a ZKP protocol in figure 7.5.

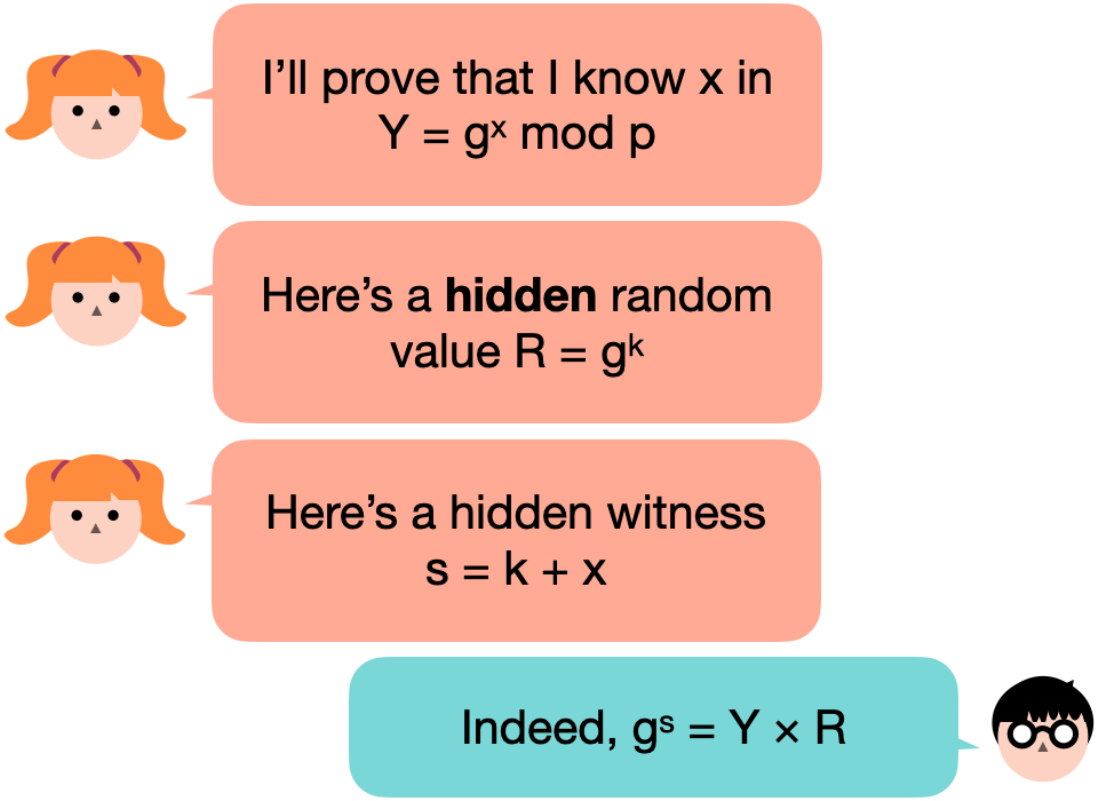


Figure 7.5 To make a knowledge proof "zero-knowledge," the prover can hide the witness x with a random value k, and hide the random value itself.

There is one last issue with our scheme: Peggy can cheat: she can convince Victor that she knows x without knowing x! All she has to do is to reverse the step in which she computes her proof: she first generates a random value s, and then calculate the value R based on s:

$R = g^s \times Y^{-1}$

Victor then computes $Y \times R = Y \times g^s \times Y^{-1}$ which indeed matches g^s. (Peggy's trick of using an inverse to compute a value is used in many attacks in cryptography.)

> **NOTE** In theoretical terms, we say that the scheme is sound if Peggy cannot cheat, meaning that if she doesn't know x then she can't fool Victor.

To make the ZKP protocol sound, Victor must ensure that Peggy computes s from R and not the inverse. To do this, Victor makes the protocol **interactive**:

1. First, Peggy must **commit** to her random value k so that she cannot change it later.
2. After receiving the commitment of Peggy, Victor introduces some of his own randomness in the protocol: he generates a random value c called a **challenge** and sends it to Peggy.
3. Finally, Peggy can compute her hidden commit based on the random value k and the challenge c.

> **NOTE** You learned about commitment schemes in chapter 2, where we used a hash function to commit to a value that we can later reveal. Commitment schemes based on hash functions do not allow us to do interesting arithmetic on the hidden value, instead we can simply raise our generator to the value: g^k (which we're already doing).

Since Peggy cannot perform the last step without Victor's challenge c, and Victor won't send that to her without seeing a commitment on the random value k, Peggy is forced to compute s based on k. The obtained protocol, which I illustrate in figure 7.6, is often referred to as the **Schnorr identification protocol**.

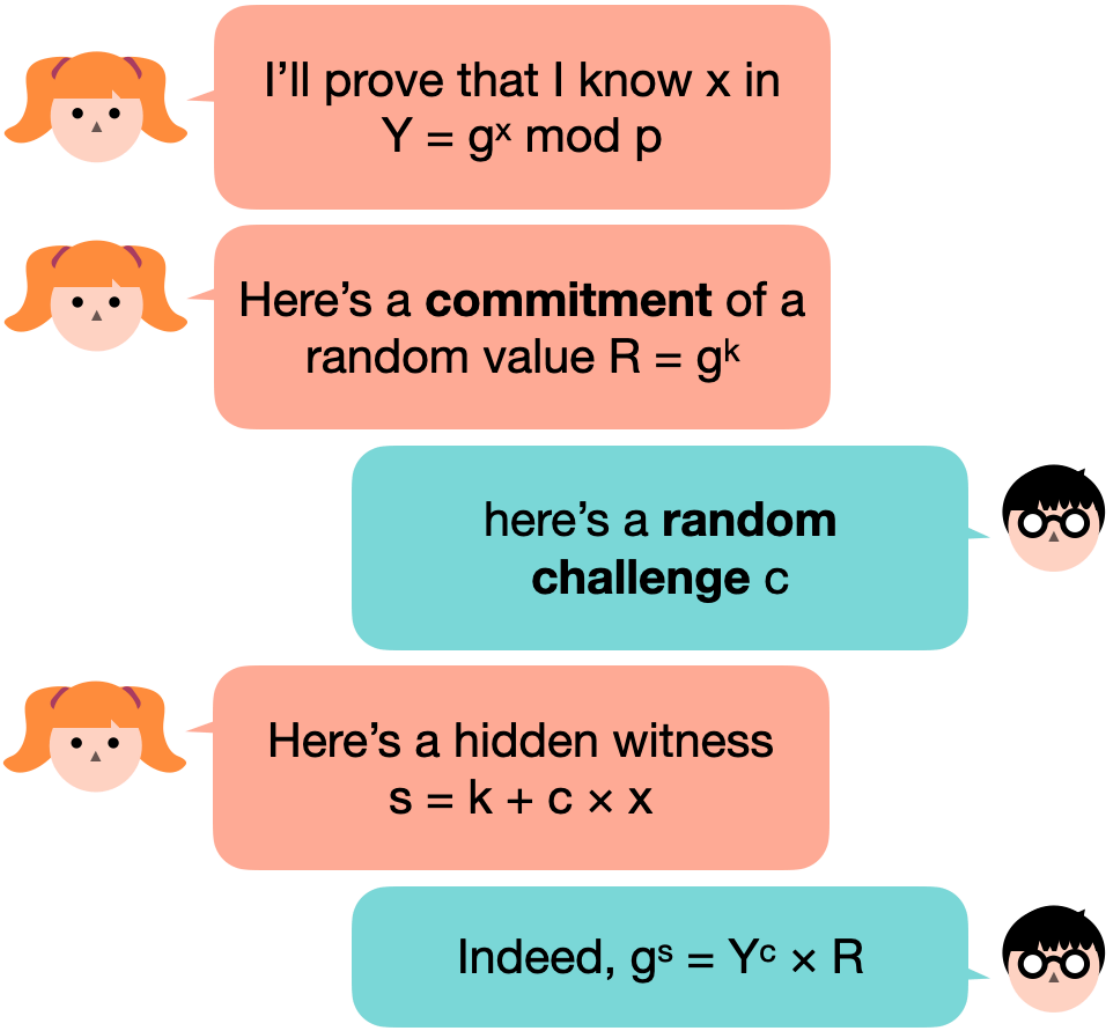


Figure 7.6 The Schnorr identification protocol is an interactive zero-knowledge proof that is complete (Peggy can prove she knows some witness), sound (Peggy cannot prove anything if she doesn't know the witness), and zero-knowledge (Victor learns nothing about the witness).

So-called **interactive zero-knowledge proof** systems which follow a three-move pattern (commitment, challenge, proof) are often referred to as **Sigma protocols** in the literature, and sometimes written as Σ-protocols due to the illustrative shape of the greek letter.

> **NOTE** The Schnorr identification protocol works in the honest verifier zero-knowledge (HVZK) model: if the verifier (Victor) acts dishonestly and does not choose a challenge randomly, they can learn something about the witness. Some stronger zero-knowledge proof schemes are zero-knowledge even when the verifier is malicious, which is often desired.

But what does that have to do with digital signatures?

7.2.2 Signatures are just non-interactive ZKPs

The problem with these interactive zero-knowledge proofs is that, well, they are **interactive**, and real-world protocols are in general not fond of interactivity. Interactive protocols add some non-negligible overheard as they require several messages (potentially over the network), and add unbounded delays unless the two participants are online at the same time. Due to this, interactive zero-knowledge proofs have been mostly absent from the world of applied cryptography.

All of this is not for nothing though! In 1986, Amos Fiat and Adi Shamir published a technique that allowed one to easily **convert an interactive ZKP into a non-interactive ZKP**. The trick they introduced (referred to as the **Fiat-Shamir heuristic** or **Fiat-Shamir transformation**) was to make the prover compute the challenge themselves, in a way they can't control.

Here's the trick: **compute the challenge as a hash of the transcript (previous messages)**. And that's it! If we assume that the hash function gives outputs that are indistinguishable from random (in other words, they look random), then it can successfully simulate a verifier.

Schnorr went a step further: he noticed that anything could be included in that hash. For example, what if we included a message in there? What we obtain is not only a proof that we know some witness *x*, but a commitment to a message that is cryptographically linked to the proof. In other words, if the proof is correct, then only someone with the knowledge of the witness (which becomes the signing key) could have committed that message.

That's a signature! Digital signatures are just non-interactive ZKPs. Applying the Fiat-Shamir transform to the Schnorr identification protocol, we obtain the **Schnorr signature** scheme which I illustrate in figure 7.7.

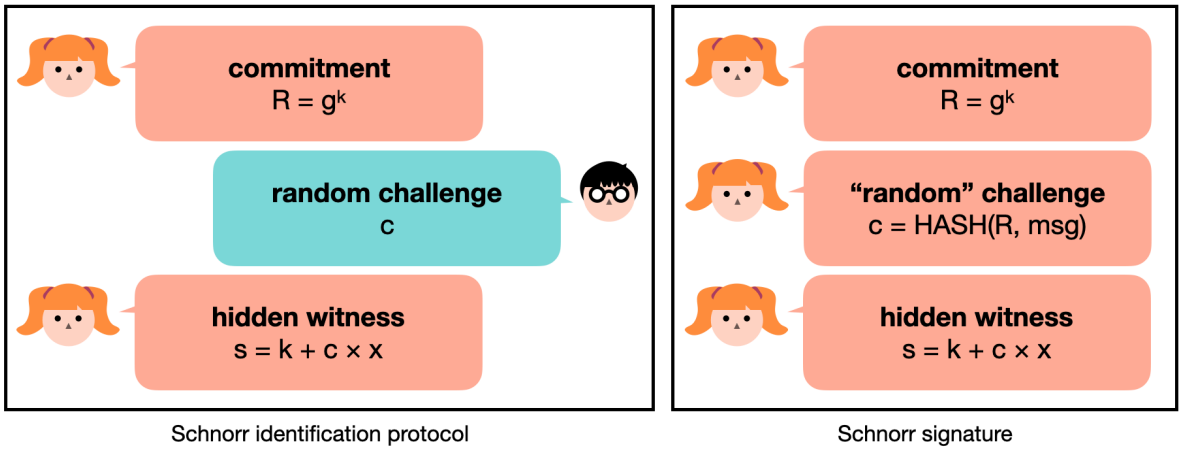


Figure 7.7 The left protocol is the Schnorr identification protocol previously discussed, which is an interactive protocol. The right protocol is a Schnorr signature, which is a non-interactive version of the left protocol (where the verifier message has been replaced by a call to a hash function on the transcript).

To recapitulate, A Schnorr signature is essentially two values R and s, where R is a commitment to some secret random value (which is often called a nonce, as it needs to be unique per-signature) and s is a value computed with the help of the commitment R, the private key (the witness x), and a message.

Next, Let's look at what the modern standards for signature algorithms are.

7.3 The signature algorithms you should use (or not)

Like other fields in cryptography, digital signatures have many standards and it is sometimes hard to understand which one to use. This is why I'm here! Fortunately, the types of algorithms for signatures are very similar to the ones for key exchanges: there are algorithms based on arithmetic modulo a large number (like Diffie-Hellman and RSA), and there are algorithms based on elliptic curves (like Elliptic Curve Diffie-Hellman). So be sure you understand the algorithms in chapter 5 and chapter 6 well enough, as we're going to build upon them.

Interestingly, the paper introducing the Diffie-Hellman key exchange (in 1976) also proposes the concept of digital signatures, without a solution:

> *In order to develop a system capable of replacing the current written contract with some purely electronic form of communication, we must discover a digital phenomenon with the same properties as a written signature. It must be easy for anyone to recognize the signature as authentic, but impossible for anyone other than the legitimate signer to produce it. We will call any such technique one-way authentication. Since any digital signal can be copied precisely, a true digital signature must be recognizable without being known.*
>
> *– Diffie and Hellman New Directions in Cryptography (1976)*

A year later in 1977, the first signature algorithm called **RSA** was introduced, along with the RSA asymmetric encryption algorithm (which you learned about in chapter 6). RSA for signing is the first algorithm we'll learn about.

In 1991, the **Digital Signature Algorithm (DSA)** was proposed by the NIST as an attempt to avoid the patents on Schnorr signatures. For this reason DSA is a weird variant of Schnorr signatures, published without a proof of security (although no attacks have been found so far). The algorithm was adopted by many, but quickly got replaced with an elliptic curve version called **ECDSA** (for Elliptic Curve DSA), the same way Elliptic Curve Diffie-Hellman (ECDH) replaced Diffie-Hellman (DH) thanks to its smaller keys (see chapter 5). ECDSA is the second signature algorithm I will talk about.

After the patents on Schnorr signatures expired in 2008, Daniel J. Bernstein, the inventor of ChaCha20-Poly1305 (covered in chapter 4) and X25519 (covered in chapter 5), introduced a new signature scheme called **EdDSA** (for Edwards-curve DSA) based on Schnorr signatures. Since its invention, EdDSA has quickly gained adoption and is nowadays considered the state of

the art in terms of digital signature for real-world applications. EdDSA is the third and last signature algorithm I will talk about in this chapter.

7.3.1 RSA PKCS#1 v1.5: a bad standard

RSA signatures are currently used everywhere, even though they shouldn't be (as you will see in this section they present many issues). This is due to the algorithm being the first signature scheme to be standardized, as well as real-world applications being really slow to move to newer and better algorithms. Because of this, you will most likely encounter RSA signatures in your journey, and I cannot avoid explaining how they work and which standards are the adopted ones. But let me say that if you understood how RSA encryption works in chapter 6, then this section should be straightforward as signing with RSA is the opposite of encrypting with RSA:

- To **sign**, you essentially encrypt the message with the private key (instead of the public key), which produces a signature (a random element in the group).
- To **verify a signature**, you essentially decrypt the signature with the public key (instead of the private key). If it gives you back the original message, then the signature is valid.

> **NOTE** **In reality, a message is often hashed before being signed, as it'll take less space (RSA can only sign messages that are smaller than its modulus). The result is also interpreted as a large number, so that it can be used in mathematical operations.**

So if your private key is the private exponent d, and your public key is the public exponent and public modulus (e, N) you:

- sign a message by computing $signature = message^d \bmod N$
- verify a signature by computing $signature^d \bmod N$ and check that it is equal to the message

I illustrated this visually in figure 7.8.

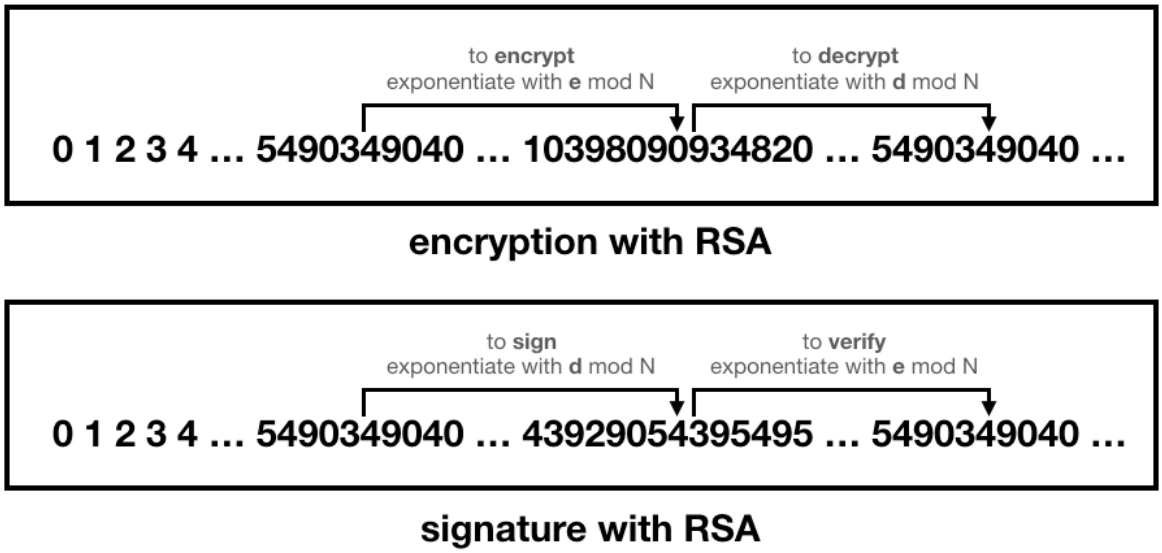


Figure 7.8 To sign with RSA, we simply do the inverse of the RSA encryption algorithm: we exponentiate the message with the private exponent, and to verify we exponentiate the signature with the public exponent (which returns back to the message).

This works because only the one knowing about the private exponent *d* can produce a signature over a message. And as with RSA encryption, the security is tightly linked with the hardness of the factorization problem.

What about the standards to use RSA for signatures? Luckily, they follow the same pattern the standards for RSA encryption did:

- RSA for encryption was loosely standardized in the **PKCS#1 v1.5** document. The same document contained a specification for signing with RSA (and without a security proof).
- RSA was then standardized again in the in PKCS#1 v2 document with a better construction called RSA-OAEP. The same happened for RSA signatures, with **RSA-PSS** (for Probabilistic Signature Scheme) being standardized in the same document (and with a security proof).

NOTE **By the way, don't get confused by the different terms surrounding RSA! There is RSA the asymmetric encryption primitive and RSA the signature primitive. (On top of that there is also RSA the company, founded by the inventors of RSA.) When mentioning encryption with RSA, most people refer to the schemes RSA PKCS#1 v1.5 and RSA-OAEP. When mentioning signatures with RSA, most people refer to the schemes RSA PKCS#1 v1.5 and RSA-PSS. I know, this can be confusing, especially for the PKCS#1 v1.5 standard. While there are official names to distinguish the encryption from the signing algorithm in PKCS#1 v1.5 –RSAES-PKCS1-v1_5 for encryption, and RSASSA-PKCS1-v1_5 for signature– I've rarely seen them used.**

I talked about RSA PKCS#1 v1.5 in chapter 6 on Asymmetric Encryption. The signature scheme standardized in that document is pretty much the same as the encryption scheme. To sign, first

hash the message with a hash function of your choice, then pad it according to PKCS#1 v1.5's padding for signatures (which is similar to the padding for encryption in the same standard) and then "encrypt" the padded and hashed message with your private exponent. I illustrated this in figure 6.14.

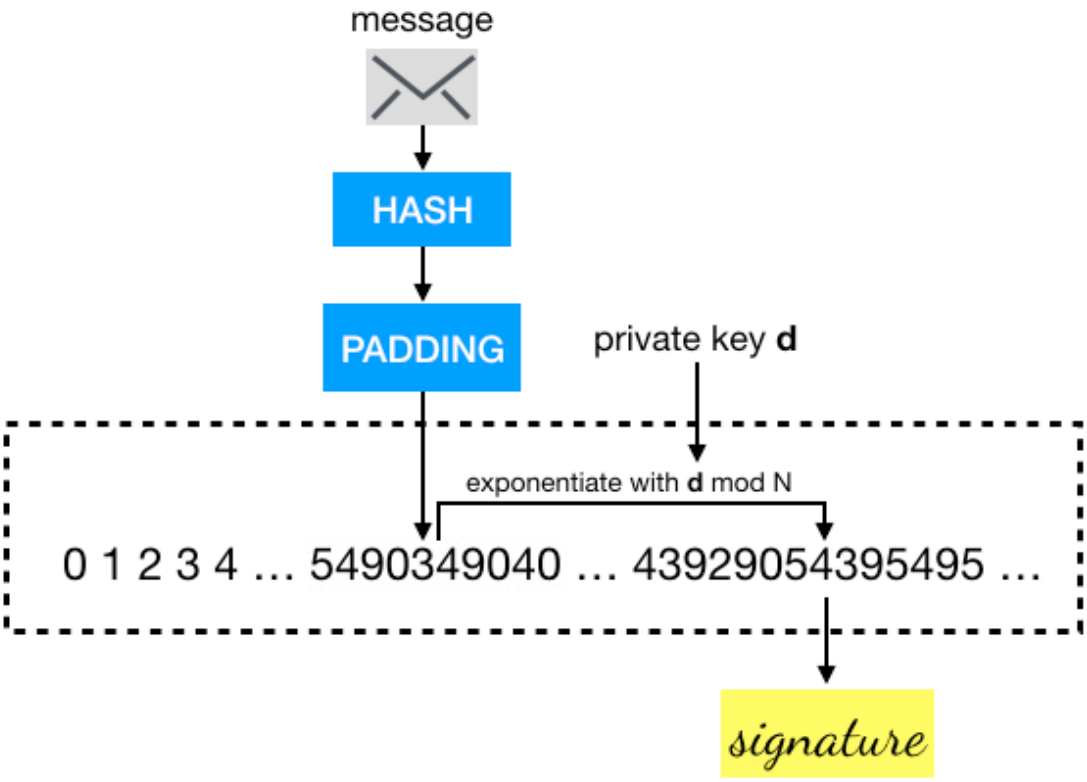


Figure 7.9 RSA PKCS#1 v1.5 for signatures. To sign, hash then pad the message with the PKCS#1 v1.5 padding scheme. The final step exponentiates the padded hashed message with the private key d modulo N. To verify, simply exponentiate the signature with the public exponent e modulo N, and verify that it matches the padded and hashed message.

In chapter 6, I mentioned that there were damaging attacks on RSA PKCS#1 v1.5 for encryption; the same is unfortunately true for RSA PKCS#1 v1.5 for signatures. In 1998, after Bleichenbacher found a devastating attack on RSA PKCS#1 v1.5 for encryption, he decided to take a look at the signature standard. Bleichenbacher came back in 2006, with a **signature forgery** attack on RSA PKCS#1 v1.5, one of the most catastrophic types of attack on signatures: attackers can forge signatures without knowledge of the private key. Unlike the first attack that broke the encryption algorithm directly, the second attack was an implementation attack. This meant that if the signature scheme was implemented correctly—according to the specification—the attack did not work.

An implementation flaw doesn't sound as bad as an algorithm flaw, that is if it's easy to avoid and doesn't impact many implementations. Unfortunately, it was shown in 2019 (see Analyzing Semantic Correctness with Symbolic Execution: A Case Study on PKCS#1 v1.5 Signature Verification by Chau et al.) that an embarassing number of open source implementations of RSA

PKCS#1 v1.5 for signatures actually fell for that trap, and mis-implemented the standard. The various implementation flaws ended up enabling different variants of Bleichenbacher's forgery attack.

Unfortunately, RSA PKCS#1 v1.5 for signatures is still widely used. So be aware of these issues if you really **have to** use this algorithm for backward compatibility reasons.

Having said that, this does not mean that RSA for signatures is insecure. The story does not end here.

7.3.2 RSA-PSS: a better standard

RSA-PSS was standardized in the updated PKCS#1 v2.1 standard, and included a proof of security (unlike the signature scheme standardized in the previous PKCS#1 v1.5). The newer specification works like this:

- Encode the message using the PSS encoding algorithm.
- Sign the encoded message using RSA (as was done in the PKCS#1 v1.5 standard).

The PSS encoding is a bit more involved, and similar to OAEP. I illustrate it in figure 7.10.

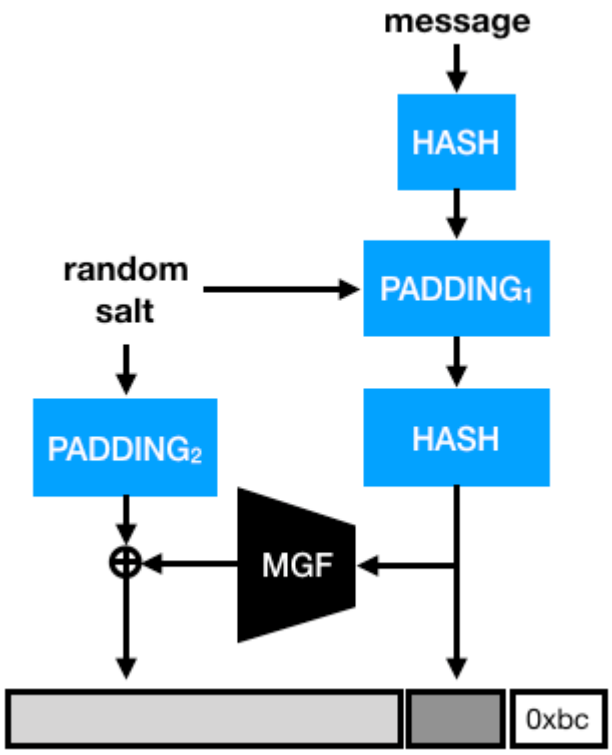


Figure 7.10 The RSA-PSS signature scheme encodes a message (using a Mask Generator Function like the RSA-OAEP algorithm you've learned about in chapter 6) before signing it in the usual RSA way.

Verifying a signature produced by RSA-PSS is just a matter of inverting the encoding once the

signature has been raised to the public exponent modulo the public modulus.

> **NOTE** PSS is provably secure (meaning that no one should be able to forge a signature without knowledge of the private key) via the usual way of proving security in asymmetric cryptography: instead of proving that if RSA (RSA-PSS without the encoding part) is secure then RSA-PSS is secure, we prove the inverse (which is equivalent): if someone can break RSA-PSS, then that someone can also break RSA. Of course this only works if RSA is indeed secure, which we assume here.

If you remember, I also talked about a third algorithm in chapter 6 for RSA encryption (called RSA-KEM). A simpler algorithm that is not used by anyone and yet is proven to be secure as well. Interestingly RSA for signatures also mirror this part of the RSA encryption history, and has a much simpler algorithm that pretty much nobody uses called **Full Domain Hash (FDH)**. FDH works by simply hashing a message and then signing it (by interpreting the digest as a number) using RSA.

Despite the fact that both RSA-PSS and FDH come with proofs of security, and are much easier to implement correctly, today most protocols still make use of RSA PKCS#1 v1.5 for signatures. This is just another example of the slowness that typically takes place around deprecating cryptographic algorithms. As older implementations still have to work with newer implementations, it is difficult to remove or replace algorithms. Think of users that do not update applications, vendors that do not provide new versions of their softwares, hardware devices that cannot be updated, and so on.

Next, let's take a look at a more modern algorithm.

7.3.3 The elliptic curve digital signature algorithm (ECDSA)

In this section let's look at the Elliptic Curve Digital Signature Algorithm (ECDSA), an elliptic curve variant of DSA which was itself invented only to circumvent patents in Schnorr signatures.

The signature scheme is specified in many standards, including ISO 14888-3, ANSI X9.62, NIST's FIPS 186-2, IEEE P1363, and so on. Not all standards are compatible, and applications that want to interoperate have to make sure that they use the same standard.

Unfortunately ECDSA, like DSA, does not come with a proof of security while Schnorr signatures did. Since then, the patents on Schnorr signatures have expired and it is making a comeback. Nonetheless, ECDSA has been widely adopted and is one of the most widely used signature schemes. In this section I will explain how ECDSA works and how it can be used.

As with all such schemes, the public key is pretty much always generated according to the same formula:

- The private key is a large randomly-generated number *x*.
- The public key is obtained by seeing *x* as an index in a group generated by a generator (called **base point** in elliptic curve cryptography).

More specifically, in ECDSA the public key is computed using *[x]G*, which is a scalar multiplication of the scalar *x* with the base point *G*.

> **NOTE** Notice that I use the additive notation (with the elliptic curve syntax of placing brackets around the scalar), but that I could have written public_key = G^x if I had wanted to use the multiplicative notation. These differences do not matter in practice. Most of the time, protocols that do not care about the underlying nature of the group are written using the multiplicative notation, whereas protocols that are defined specifically in elliptic curve-based groups tend to be written using the additive notation.

To compute an ECDSA signature, you need the same inputs required by a Schnorr signature: a hash of the message you're signing *H(m)*, your private key *x*, a random number *k* unique per-signature. An ECDSA signature is two integers r and s computed as follows:

- *r* is the x-coordinate of *[k]G*
- $s = k^{-1} (H(m) + x\ r) \bmod p$

To verify an ECDSA signature, a verifier needs to use the same hashed message *H(m)*, the signer's public key and the message signature *(r,s)*:

- Compute $[H(m)\ s^{-1}]G + [r\ s^{-1}]public\_key$.
- The signature is valid if the x-coordinate of the point obtained is the same as the value *r* of the signature.

You can certainly recognize that there are some similarities with Schnorr signatures. The random number *k* is sometimes called a **nonce**, because it is a number that must only be used once, and is sometimes called an **ephemeral key** because it must remain secret.

> **WARNING** I'll reiterate this: k must never be repeated nor be predictable! Without that, it is trivial to recover the private key.

In general, cryptographic libraries will hide the generation of this nonce and do it behind the scenes, but sometimes they don't and let the caller provide it. This is of course a recipe for disaster. For example, Sony's Playstation 3 was found using ECDSA with repeating nonces in 2010 (which leaked their private keys).

> **NOTE** Even more subtle, if the nonce k is not totally picked uniformly and at random (specifically, if you can predict the first few bits), there still exist very powerful attacks that can recover the private key in no time (so-called lattice attacks). In theory, we call these kinds of key retrieval attacks total breaks (because they break everything)! Such total breaks are quite rare in practice, which makes ECDSA an algorithm that can fail in spectacular ways.

Attempts at avoiding issues with nonces exist. For example, RFC 6979 specifies a **deterministic ECDSA** scheme which generates a nonce based on the message and the private key. This means that signing the same message twice involves the same nonce twice, and as such produces the same signature twice (which is obviously not a problem).

The elliptic curves that tend to be used with ECDSA are pretty much the same curves that are popular with the Elliptic Curve Diffie-Hellman algorithm (see chapter 5), with one notable exception: **Secp256k1**.

The Secp256k1 curve is defined in the SEC 2 standard (Recommended Elliptic Curve Domain Parameters) written by the Standards for Efficient Cryptography Group (SECG). It has gained a lot of traction after Bitcoin decided to use it instead of the more popular NIST curves, due to the lack of trust in the NIST curves I've mentioned in chapter 5.

Secp256k1 is a type of elliptic curve called a Koblitz curve. A Koblitz curve is just an elliptic curve with some constraints in its parameters, which allow implementations to optimize some operations on the curve. The elliptic curve has the following equation:

$y^2 = x^3+ax+b$

With a=0 and b=7 constants, and with x and y defined over the numbers modulo the prime p:

$p = 2^{192} - 2^{32} - 2^{12} - 2^8 - 2^7 - 2^6 - 2^3 - 1$

This defines a group of prime order, like the NIST curves. Today, we have efficient formulas to compute the number of points on an elliptic curve, so here is the prime number that is the number of points in the Secp256k1 curve (including the point at infinity):

115792089237316195423570985008687907852837564279074904382605163141518161494337

And we use as generator (or base point) the fixed point G of coordinates x = *55066263022277343669578718895168534326250603453777594175500187360389116729240* and y = *32670510020758816978083085130507043184471273380659243275938904335757337482424*.

Nonetheless, today ECDSA is mostly used with the NIST curve P-256 (sometimes referred to as

Secp256r1, note the difference).

Next let's look at another widely popular signing scheme.

7.3.4 The Edwards-curve digital signature algorithm (EdDSA)

Let me introduce the last signature algorithm of the chapter, The **Edwards-curve Digital Signature Algorithm (EdDSA)**, published in 2011 by Daniel J. Bernstein in response to the lack of trust in NIST and other curves created by government agencies.

The name EdDSA seems to indicate that it is based on the DSA algorithm, like ECDSA is, but this is deceiving. EdDSA is actually based on Schnorr signatures, which has been possible due to the patent on Schnorr signatures expiring earlier in 2008.

One particularity of EdDSA is that the scheme does not require new randomness for every signing operation. EdDSA produces signatures **deterministically**. This has made the algorithm quite attractive, and it has since been adopted by many protocols and standards.

EdDSA is on track to be included in NIST's upcoming update for its FIPS 186-5 standard (still a draft as of early 2021). The current official standard is RFC 8032, which defines two curves of different security levels to be used with EdDSA. Both of the defined curves are twisted Edwards curves (a type of elliptic curve enabling interesting implementation optimizations):

- **Edwards25519** is based on Daniel J. Bernstein's Curve25519 (covered in chapter 5). Its curve operations can be implemented faster than those of Curve25519, thanks to the optimizations enabled by the type of elliptic curve. As it was invented after Curve25519, the key exchange X25519 based on Curve25519 did not benefit from these speed improvements. As with Curve25519, Edwards25519 provides 128-bit of security.
- **Edwards448** is based on Mike Hamburg's Ed448-Goldilocks curve, providing 224-bit of security.

In practice, EdDSA is mostly instantiated with the Edwards25519 curve, and the combo is called **Ed25519** (whereas EdDSA with Edwards448 is shortened as Ed448).

Key generation with EdDSA is a bit different from other existing schemes. Instead of generating a signing key directly, EdDSA generates a secret key that is then used to derive the actual signing key and another key we'll call the nonce key. That nonce key is important, it is the one used to deterministically generate the required per-signature nonce.

> **NOTE** **Depending on the cryptographic library you're using, you might be storing the secret key, or the two derived keys: the signing key and the nonce key. Not that this matters, but if you don't know this you might get confused if you run into Ed25519 secret keys being stored as 32 bytes and 64 bytes depending on the implementation used.**

To sign, EdDSA first deterministically generates the nonce by hashing the nonce key with the message to sign. After that, a process similar to Schnorr signatures follows:

1. compute the nonce as *HASH(nonce key || message)*
2. compute the commitment *R* as *[nonce]G* with *G* the base point of the group.
3. compute the challenge as *HASH(commitment || public key || message)*
4. compute the proof *S* as *nonce + challenge × signing_key*
5. the signature is *(R, S)*

I illustrate the important parts of EdDSA in figure 7.11.

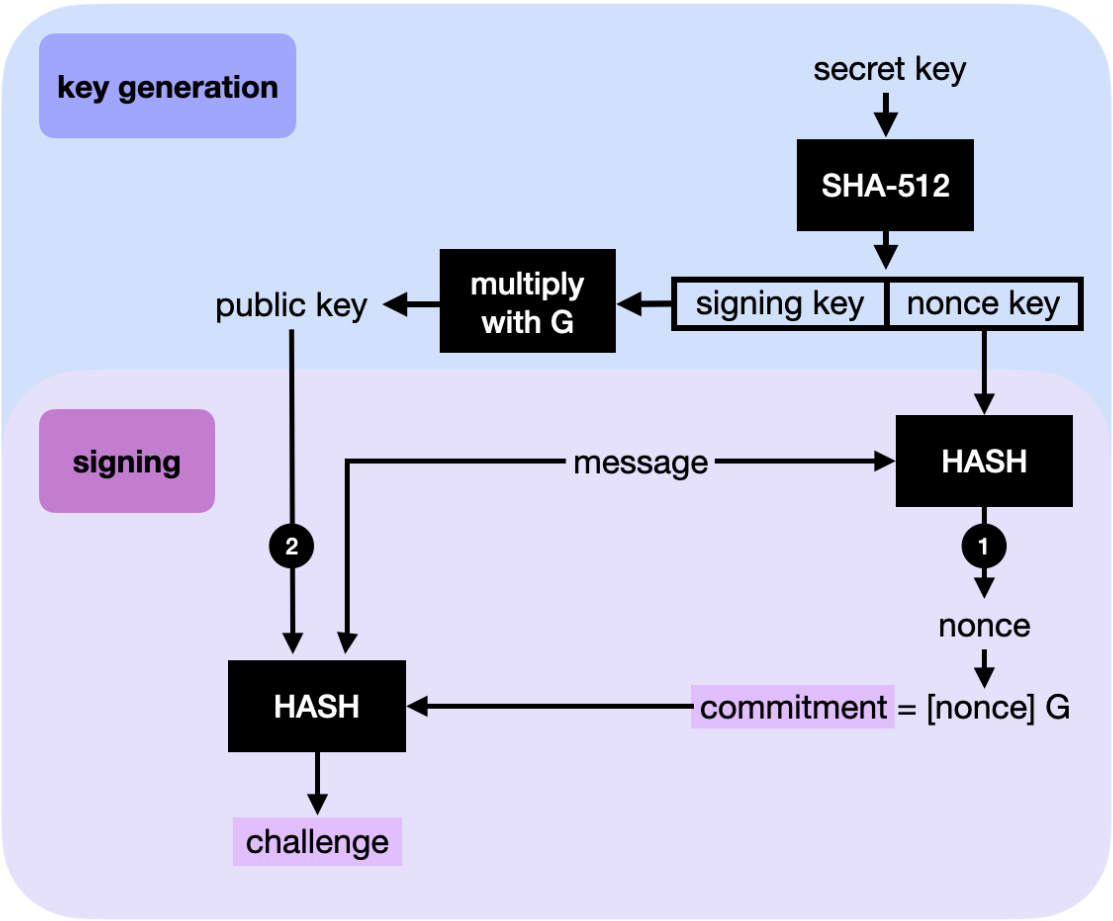


Figure 7.11 EdDSA key generation produces a secret key that is then used to derive two other keys. The first derived key is the actual signing key, and can thus be used to derive the public key; the other derived key is the nonce key, used to deterministically derive the nonce during signing operations. EdDSA signatures are then like Schnorr signatures, for the exception that (1) the nonce is generated deterministically from the nonce key and the message, and (2) the public key of the signer is included as part of the challenge.

Notice how the nonce (or ephemeral key) is derived deterministically, and not probabilistically, from the nonce key and the given message. This means that signing two different messages should involve two different nonces, ingeniously preventing the signer from reusing nonces, and in turn leaking out the key (as you've seen can happen with ECDSA). Signing the same message

twice will produce the same nonce twice, producing the same signature twice as well. This is obviously not a problem.

A signature can be verified by computing the following two equations:

- *[S]G*
- *R + [HASH(R || public key || message)] public key*

The signature is valid if the two values match. This is exactly how Schnorr signatures work, except that we are now in an elliptic curve group, and I used the additive notation.

The most-widely used instantiation of EdDSA, **Ed25519**, is defined with the Edwards25519 curve and the SHA-512 as hash function. The Edwards25519 curve is defined with all the points satisfying this equation

$-x^2 + y^2 = 1 + d \times x^2 \times y^2 \bmod p$

where the value d is the large number 37095705934669439343138083508754565189542113879843219016388785533085940283555; and the variables x and y are taken modulo p the large number 2^{255} - 19 (the same prime used for Curve25519).

The base point is G of coordinate x = 15112221349535400772501151409588531511454012693041857206046113283949847762202 and y = 46316835694926478169428394003475163141307993866256225615783033603165251855960

RFC 8032 actually defines 3 variants of EdDSA using the Edwards25519 curve. All 3 variants follow the same key generation algorithm, but with different signing and verification algorithms:

- **Ed25519 (or pureEd25519)**. That's the algorithm that I've explained above
- **Ed25519ctx**. This introduces a mandatory customization string. This algorithm is rarely implemented, if even used, in practice. The only difference is that some user-chosen prefix is added to every call to the hash function.
- **Ed25519ph (or HashEd25519)**. This allows applications to pre-hash (hence the "ph" in the name) the message before signing it. It also builds upon Ed25519ctx, allowing the caller to include an optional custom string.

The addition of a **customization string** is quite common in cryptography, as you have seen with some hash functions in chapter 2, or key derivation functions in chapter 8. It is a useful addition when a participant in a protocol uses the same key to sign messages in different context. For example, you can imagine an application that would allow you to sign transactions using your private key, but also to sign private messages to people you talk to. If you mistakenly sign and

send a message to your evil friend Eve that looks like a transaction, she could try to re-publish it as a valid transaction, that is if there's no way to distinguish the two types of payload you're signing.

Ed25519ph was introduced solely to please callers that need to sign large messages: as you've seen in chapter 2, hash functions often provide an "init-update-finalize" interface that allows you to continuously hash a stream of data without having to keep the whole input in memory.

You are now done with your tour of the signature schemes used in real-world applications.

To recap:

- **RSA PKCS#1 v1.5** is still widely in use, but is hard to implement correctly and many implementations have been found to be broken.
- **RSA-PSS** has a proof of security, is easier to implement, but has seen poor adoption due to newer schemes based on elliptic curves.
- **ECDSA** is the main competition to RSA PKCS#1 v1.5, it is mostly used with NIST's curve P-256, except in the cryptocurrency world where Secp256k1 seems to dominate.
- **Ed25519** is based on Schnorr signatures and has received wide adoption. It is easier to implement compared to ECDSA, and does not require new randomness for every signing operation. This is the algorithm you should use if you can.

Next, let's look at how you can possibly shoot yourself in the foot when using these signature algorithms.

7.4 Subtle behaviors of signature schemes

There are a number of subtle properties that signature schemes might exhibit. While they might not matter in most protocols, not being aware of these gotchas can end up biting you when working on more complex and non-conventional protocols. The end of this chapter will focus on known issues with digital signatures.

A digital signature does not uniquely identify a key or a message
– Andrew Ayer Duplicate Signature Key Selection Attack in Let's Encrypt (2015)

Substitution attacks, also referred to as **Duplicate Signature Key Selection (DSKS)**. Substitution attacks are possible on both RSA PKCS#1 v1.5 and RSA-PSS. There are two variants that exist:

1. **key substitution** attacks, where a **different keypair or public key** is used to validate a given signature over a given message.
2. **message key substitution** attacks, where a **different keypair or public key** is used to validate a given signature over a **new** message.

One more time: the first attack fixes both the message and the signature, the second one only

fixes the signature. I recapitulate this in figure 7.12.

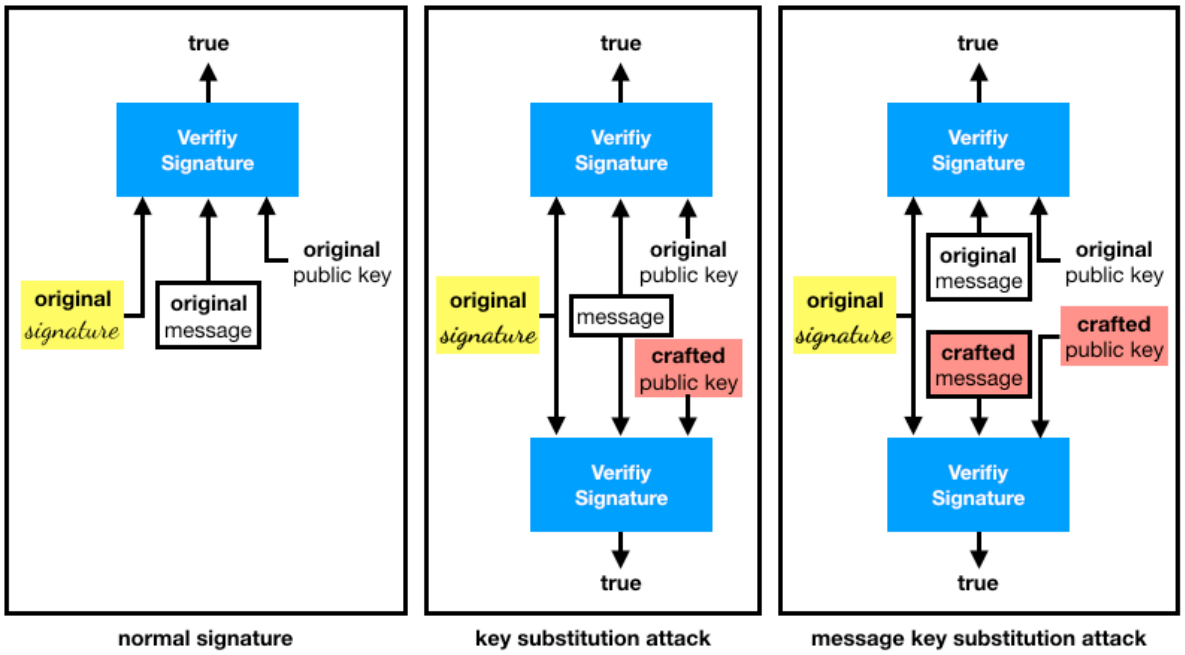


Figure 7.12 Signature algorithms like RSA are vulnerable to key substitution attacks, which are a surprising and unexpected behavior for most users of cryptography. A key substitution attack allows one to take a signature over a message, and to craft a new key pair that will validate the original signature. A variant called message key substitution allows an attacker to create a new key pair and a new message that validates under the original signature.

> **NOTE** Substitution attacks are a syndrom of a gap between theoretical cryptography and applied cryptography. Signatures in cryptography are usually analyzed with the EUF-CMA model, which stands for Existential Unforgeability under Adaptive Chosen Message Attack. In this model you generate a key pair, and then I request you to sign a number of arbitrary messages. While I observe the signatures you produce, I win if I can at some point in time produce a valid signature over a message I hadn't requested. Unfortunately, this EUF-CMA model doesn't seem to encompass every edge cases, and dangerous subtleties like the substitution ones are not taken into accounts.

In February 2014 MtGox, once the largest Bitcoin exchange, closed and filed for bankruptcy claiming that attackers used malleability attacks to drain its accounts.

– Christian Decker and Roger Wattenhofer Bitcoin Transaction Malleability and MtGox (2014)

Malleability. Most signature schemes are malleable: if you give me a valid signature, I can modify the signature so that it becomes a different—but still-valid—signature. I have no clue what the signing key was, yet I managed to create a new valid signature.

NOTE **A newer security model called SUF-CMA (for strong EUF-CMA) attempts to include non-malleability (or resistance to malleability) in the security definition of signature schemes. Some recent standards, like RFC 8032 which specifies Ed25519, include mitigations against malleability attacks. Since these mitigations are not always present, nor common, you should never rely on signatures being non-malleable in your protocols.**

Non-malleability does not necessarily mean that signatures are unique: if I'm the signer, I can usually create different signatures for the same message, and that's usually OK. Some constructions like verifiable random functions (that you'll see later in chapter 8) rely on signature uniqueness and so they must deal with this, or use signature schemes that have unique signatures (like the Boneh–Lynn–Shacham, or BLS, signatures).

What to do with all of this information? Rest assured, signature schemes are definitely not broken, and you probably shouldn't worry if your use of them is not too out-of-the-box. But if you're designing cryptographic protocols, or if you're implementing a protocol that's more complicated than the everyday cryptography, you might want to keep these subtle properties in the back of your mind.

7.5 Summary

- Digital signatures are similar to pen-and-paper signatures but are backed with cryptography, making them unforgeable by anyone who does not control the signing (private) key.
- Digital signatures can be useful to authenticate origins (for example, one side of a key exchange) as well as providing transitive trust (if I trust Alice and she trusts Bob, I can trust Bob).
- Zero-knowledge proofs (ZKPs) allow a prover to prove the knowledge of a particular piece of information (called a witness), without revealing the something. Signatures can be seen as non-interactive zero-knowledge proofs as they do not require the verifier to be online during the signing operation.
- You can use many standards to sign:
    - RSA PKCS#1 v1.5 for signing is widely used today, but not recommended as it is hard to implement correctly.
    - RSA-PSS is a better signature scheme, as it is easier to implement and has a proof of security. Unfortunately it is not very popular nowadays due to elliptic curve variants that support shorter keys and are thus more attractive for network protocols.
    - The most popular signature schemes right now are based on elliptic curves: ECDSA and EdDSA. ECDSA is often used with NIST curve P-256 and EdDSA is often used with the Edwards25519 curve and the combination is referred to as Ed25519.
- Some subtle properties can be dangerous if signatures are used in a nonconventional way:
    - always avoid ambiguity as to who signed a message as some signature schemes are vulnerable to key substitution attacks: external actors can create a new keypair that would validate an already existing signature over a message, or create a new keypair and a new message that would validate a given signature.
    - do not rely on uniqueness of signatures. First, in most signature schemes the signer can create an arbitrary amount of signatures for the same message. Second, most signature schemes are "malleable," meaning that external actors can take a signature and create another valid signature for the same message.

8 Randomness and secrets

This chapter covers

- What randomness is and why it's important.
- Obtaining strong randomness and producing secrets for cryptography.
- What the pitfalls of randomness are.

This is the last chapter of the first part of this book, and I have one last thing to tell you before we move on to the second part of this book and learn about actual protocols used in the real-world. It is something I've grossly negliged so far: **randomness**.

You must have noticed that in every cryptographic algorithm you've learned, with the exception of hash functions, you had to use randomness at some point. Secret keys, nonces, IVs, prime numbers, challenges, and so on. As I was going through these different concepts, randomness always came from some magic black box. This is not atypical, in cryptography whitepapers randomness is often represented by drawing an arrow with a dollar sign on top. But at some point, we do need to ask ourselves the question: where does this randomness really come from?

In this chapter, I will provide you with an answer as to what cryptography means when it mentions randomness, and I will give you pointers about the practical ways that exist to obtain randomness for real-world cryptographic applications.

For this chapter you'll need to have read:

- Chapter 2 on hash functions.
- Chapter 3 on message authentication codes.

8.1 What's randomness?

Everyone understands the concept of randomness to some degree. Whether playing with dice or buying some lottery tickets, we've all been exposed to it. My first encounter with randomness was at a very young age, when I realized that a "RAND" button on my calculator would produce a different number every time I would press it. This troubled me to no end. I had very little knowledge about electronics, but I thought I could understand some of its limitations. When I added 4 and 5 together, surely some circuits would do the math and give me the result. But a random button? Where were the random numbers coming from? I couldn't wrap my head around it. It took me some time to ask the right questions and understand that calculators actually cheated! They would hardcode large lists of "random" numbers and go through them one by one. These lists would exhibit good randomness, meaning that if you looked at the random numbers you were getting, there'd be as many 1s as 9s, and as many 1s as 2s, and so on. These lists would simulate a **uniform distribution**: the numbers were distributed in equal proportions (uniformly).

When random numbers are needed for security and cryptography purposes, then randomness must be **unpredictable**. So of course, nobody should have used these calculators' randomness for anything related to security. Instead, cryptographic applications **extract** randomness from observing **hard-to-predict physical phenomena**.

For example, it is hard to predict the outcome of a dice roll, even though throwing a die is a deterministic process; if you knew all the initial conditions (how you're throwing the die, the die itself, the air friction, the grip of the table, and so on) you should be able to predict the result. That being said, all of these factors impact the end product so much that a slight imprecision in the knowledge of the initial conditions would mess with our predictions. The extreme sensitivity of an outcome to its initial conditions is known as **chaos theory**, and it is the reason why things like the weather are extremely hard to predict accurately past a certain number of days.

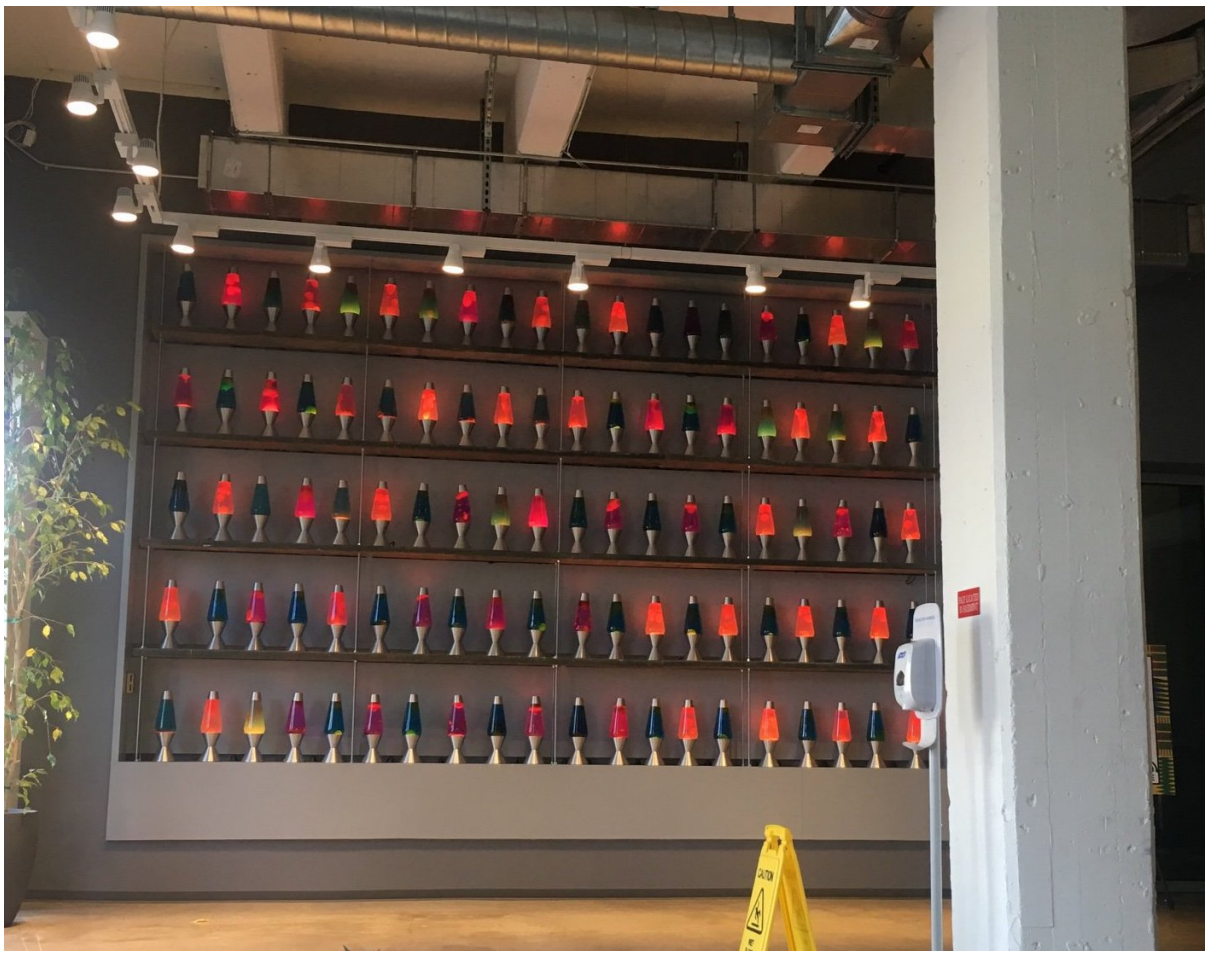


Figure 8.1 A picture I snapped during one of my visits to the headquarters of Cloudflare in San Francisco. LavaRand is a wall of lava lamp, which are lamps that produce hard-to-predict shapes of wax. A camera is set in front of the wall to extract and convert these images to random bytes.

Applications usually rely on the operating system (OS) to provide usable randomness, who in turn gather randomness using different tricks depending on the type of device it is being ran on. Common sources of randomness (also called **entropy sources**) can be the timing of hardware interrupts (for example, your mouse movements), software interrupts, hard disk seek time, and so on.

NOTE In information theory the word entropy is used to judge how much randomness a string contains. The term was coined by Claude Shannon who devised an entropy formula that would output larger and larger numbers as a string would exhibit more and more unpredictability (starting at 0 for completely predictable). The formula or the number itself is not that interesting for us, but in cryptography you will often hear "this string has low entropy" (meaning that it is predictable) or "this string has high entropy" (meaning that it is less predictable).

Observing interrupts and other events to produce randomness is not great: when a device boot

these events tend to be highly predictable, and they can also be maliciously influenced by external factors. Today, more and more devices have access to additional sensors and hardware aids that can provide better sources of entropy. These hardware random number generators are often called **true random number generators (TRNGs)** as they make use of external unpredictable physical phenomena (like thermal noise) to extract randomness.

The noise obtained via all these different types of input is usually not "clean" and can sometimes not provide enough entropy (if at all). For example, the first bit obtained from some entropy source could be 0 more often than not, or successive bits could be more likely than chance to be equal. Due to this, randomness extractors must clean and gather several sources of noise together before it can be used for cryptographic applications. This can be done, for example, by applying a hash function to the different sources and XORing the digests together.

Is this all there is to randomness? Unfortunately not. Extracting randomness from noise is a process that can be slow. For some applications that might need lots of random numbers very quickly, it can become a bottleneck.

The next section will go over how Os's and real-world applications boost the generation of random numbers.

8.2 Slow randomness? use a pseudo-random number generator (PRNG)

Randomness is used everywhere. At this point you should be convinced that this is true at least for cryptography, but surprisingly cryptography is not the only place making heavy use of random numbers. For example, simple Unix programs like `ls` require randomness too! As a bug in a program can be devastating if exploited, binaries attempt to defend against low-level attacks using a multitude of tricks; one of them is ASLR, which randomizes the memory layout of a process every time it is run (and thus requires random numbers). Another example is the network protocol TCP, which makes use of random numbers every time it creates a connection to produce unpredictable sequence numbers and thwart attacks attempting to hijack connections. While all of this is beyond the scope of this book, it is good to have an idea of how much randomness ends up being used for security reasons in the real world.

Unfortunately, I hinted in the last section that obtaining unpredictable randomness is somewhat of a slow process (this is sometimes due to a source of entropy being slow to produce noise). As a result, OS's often optimize their production of random numbers by using **pseudo-random number generators (PRNGs)**.

> **NOTE** PRNGs are sometimes shortened as PRGs, or sometimes called CSPRNG for "cryptographically secure" PRNGs in order to contrast with PRNGs not designed to be secure (useful in video games, for example). The NIST, wanting to do things differently than everybody else (as usual), often calls theirs deterministic random bit generators (DRBGs).

A PRNG needs an initial secret, usually called a **seed** (which we can obtain from mixing different entropy sources together), and can then produce lots of random numbers very quickly. I illustrate a PRNG in figure 8.2.

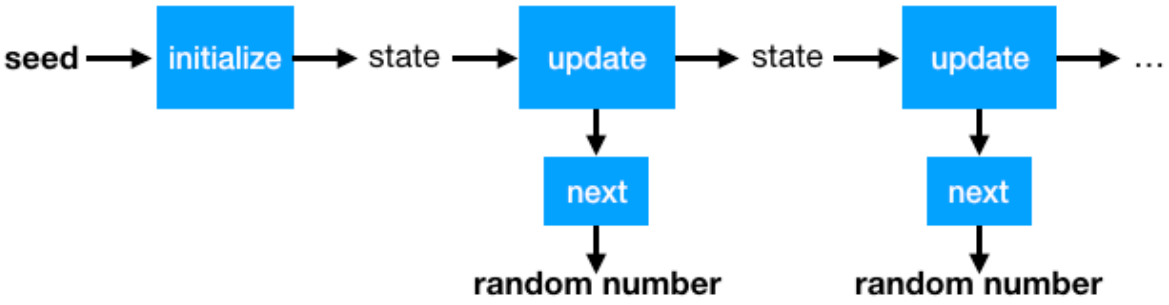


Figure 8.2 A Pseudo-Random Number Generator (PRNG) generates a sequence of random numbers based on a seed. Using the same seed makes the PRNG produce the same sequence of random numbers. It should be impossible to recover the state using knowledge of the random outputs (the function `next` is one-way). It follows that it should be impossible, from observing the produced random numbers alone, to predict future random numbers or to recover previously generated random numbers.

Cryptographically secure PRNGs usually tend to exhibit the following properties:

- **deterministic**. Using the same seed twice will produce the same sequence of random numbers. This is unlike the unpredictable randomness extraction I talked about previously: if you know a seed used by a PRNG, it should be completely predictable. This is why the construction is called **pseudo**-random, and this is what allows a PRNG to be extremely fast.
- **indistinguishability from randomness**. In practice, this means that you should not be able to distinguish between a PRNG outputting a random number from a set of possible numbers, and a little fairy impartially choosing a random number from the same set (assuming the fairy knows a magical way to pick a number such that every possible number can be picked with equal probability). Consequently, observing the random numbers generated alone shouldn't allow anyone to recover the internal state of the PRNG.

The last point is very important, as a PRNG simulates picking a number **uniformly at random**, meaning that each number from the set has an equal chance of being picked. For example, if your PRNG produces random numbers of 8 bytes, the set is all the possible strings of 8 bytes,

and each 8-byte value should have equal probability of being the next value that can be obtained from your PRNG. This includes values that have already been produced by the PRNG at some point in the past.

In addition, many PRNGs exhibit additional security properties.

A PRNG has **forward secrecy** if an attacker learning the state (by getting in your computer at some point in time, for example) doesn't allow them to retrieve previously-generated random numbers. I illustrate this in figure 8.3.

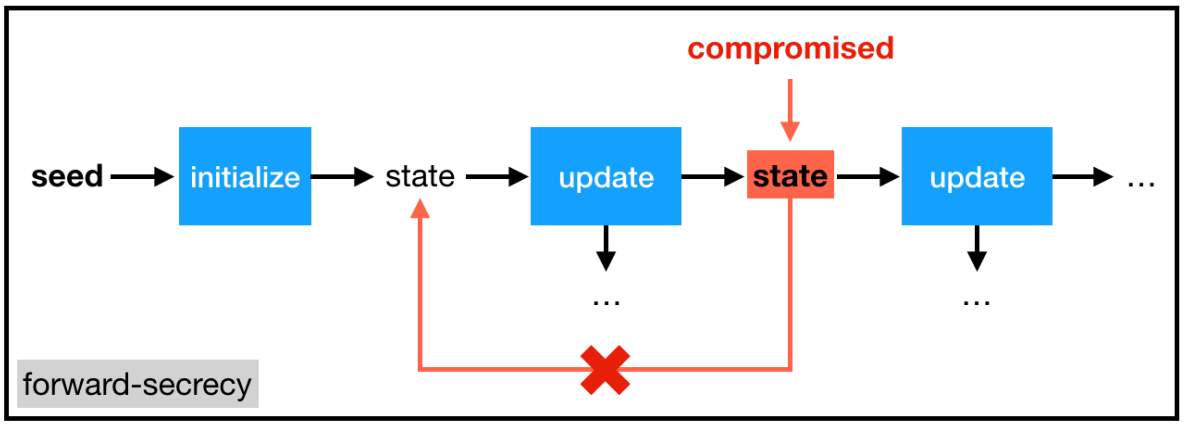


Figure 8.3 A Pseudo-Random Number Generator (PRNG) has forward secrecy if compromise of a state does not allow recovering previously generated random numbers.

Obtaining the state of a PRNG means that you can determine all future pseudo-random numbers it will generate. To prevent this, some PRNGs have mechanisms to "heal" themselves if compromised. This healing can be achieved by reinjecting (or re-seeding) new entropy (after a PRNG was already seeded). This property is called **backward secrecy**. I illustrate this in figure 8.4.

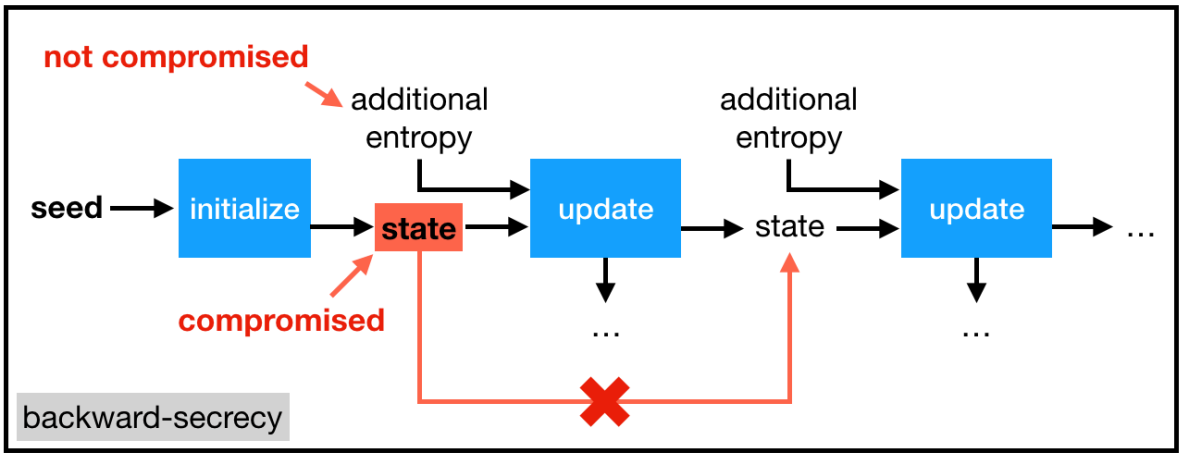


Figure 8.4 A Pseudo-Random Number Generator (PRNG) has backward secrecy if compromise of a state does not allow predicting future random numbers that will be generated by the PRNG. This is of course only true once new entropy has been produced and injected in the update function after the compromise.

> **NOTE** The terms forward and backward secrecy are often a source of confusion. If you read this section thinking "shouldn't forward secrecy be backward secrecy, and backward secrecy be forward secrecy instead?" then you are not crazy. For this reason, backward secrecy is sometimes called future-secrecy, or even post-compromise security.

PRNGs can be extremely fast, and are considered sure methods to generate large numbers of random values for cryptographic purposes, that is if properly seeded. Using a predictable number, or a number that is too small, is obviously not a secure way to seed a PRNG. This effectively means that we have secure cryptographic ways for quickly stretching a secret (of appropriate size) to billions of other secret keys. Pretty cool, right?

This is why most (if not all) cryptographic applications do not use random numbers directly extracted from noise, but instead use them to seed a PRNG in an initial step, and then switch to generating random numbers from the PRNG when needed.

> **NOTE** Today, PRNGs are mostly heuristic-based constructions. This is because constructions based on hard mathematical problems (like the discrete logarithm) are too slow to be practical. One notorious example is Dual EC, which was invented by the NSA and relied on elliptic curves. The PRNG was pushed to various standards (including NIST ones) around 2006, and not too long after several researchers independently discovered a potential backdoor in the algorithm. This was later confirmed by the Snowden revelations in 2013, and a year later the algorithm was withdrawn from multiple standards.

To be secure, a PRNG must be seeded with an unpredictable secret. More accurately, we say that the PRNG takes a key of n bytes sampled **uniformly at random**, meaning that we should pick

the key randomly from the set of all possible n-byte strings where each bytestring has the same chance of being picked.

In this book I've talked about many cryptographic algorithms that produce outputs indistinguishable from randomness (from values that would be chosen uniformly at random). Intuitively, you should be thinking "can we use these algorithms to generate random numbers then?" And you would be right! Hash functions, XOFs, block ciphers, stream ciphers, and MACs can be used to produce random numbers. Hash functions and MACs are theoretically not defined as providing outputs that are indistinguishable from random, but in practice they often are. Asymmetric algorithms like key exchange and signatures, on the other hand, are almost all the time not indistinguishable from random. For this reason, their output is often hashed before being used as random numbers.

Actually, since AES is hardware supported on most machines, it is customary to see AES-CTR being used to produce random numbers: the symmetric key becomes the seed, and the ciphertexts becomes the random numbers (for the encryption of an infinite string of zeros, for example). In practice, there is a bit more complexity added to these constructions in order to provide forward and backward secrecy.

You now understand enough to go to the next section, which will provide an overview of obtaining randomness for real.

8.3 Obtaining randomness in practice

You've learned about the three ingredients that an OS needs to provide cryptographically secure random numbers to its programs:

1. **Noise sources**. Ways for the OS to obtain raw randomness from impredictable physical phenomena like the temperature of the device or your mouse movements.
2. **Cleaning and mixing**. Since raw randomness can be of poor quality (some bits might be biased), OS's clean up and mix a number of sources together in order to produce a good random number.
3. **PRNGs**. Since the first two steps are slow, a single uniformly random value can be used to seed a PRNG which will produce a (practically) infinite number of random numbers very quickly.

In this section I will explain how systems bundle these three concepts together to provide simplified interfaces to developers. These functions exposed by OS's usually allow you to generate a random number by calling a system call. Behind these sytem calls, is indeed a system bundling up noise sources, a mixing algorithm and a PRNG (summarized in figure 8.5).

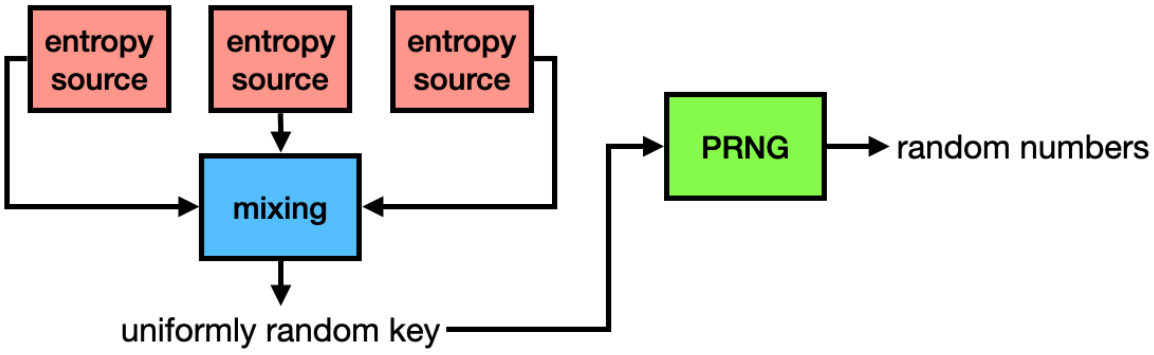


Figure 8.5 Generating random numbers on a system usually means that entropy was mixed together from different noise sources and used to seed a long-term PRNG.

Depending on the OS, and on the hardware available, these three concepts might be implemented differently. In 2021, Linux uses a PRNG based on the ChaCha20 stream cipher while macOS uses a PRNG based on the SHA-1 hash function. In addition, the random number generator interface exposed to developers will be different depending on the OS. On Windows, the `BCryptGenRandom` system call can be used to produce random numbers, while on other platforms a special file usually called `/dev/urandom` is exposed and can be read to provide randomness. For example, on Linux or macOS, one can read 16 bytes from the terminal using the `dd` command line tool:

```
$ dd if=/dev/urandom bs=16 count=1 2> /dev/null | xxd -p
40b1654b12320e2e0105f0b1d61e77b1
```

One problem with `/dev/urandom` is that it might not provide enough entropy (its numbers won't be random enough) if used too early after booting the device. Operating systems like **Linux** and **FreeBSD** offer a solution called `getrandom`, which is a system call that pretty much offers the same functionnality as reading from `/dev/urandom`; but in the rare cases where not enough entropy is available for initializing its PRNG, `getrandom` will block until properly seeded. For this reason, I recommend that you use `getrandom` if available on your system.

The following example shows how one can securely use `getrandom` in C:

Listing 8.1 random_numbers.c

```
#include <sys/random.h>

uint8_t secret[16];  ❶
int len_read = getrandom(secret, sizeof(secret), 0);  ❷

if (len_read != sizeof(secret)) {
    abort();  ❸
}
```

❶ We want to fill a buffer with random bytes (note that `getrandom` is limited to up to 256 bytes per call).

❷ `getrandom` with no flags (`0`) defaults to not blocking unless it has not been properly seeded yet.

❸ It is possible that the function fails or return less than the desired amount of random bytes. If this is the case, the system is corrupt and aborting might be the only good thing to do.

With that example in mind, it is also good to point out that many programming languages have standard libraries and cryptographic libraries that provide better abstractions. It might be easy to forget that `getrandom` will only return up to 256 bytes per call, for example. For this reason, you should always attempt to generate random numbers through the standard library of the programming language you're using.

WARNING **Note that many programming languages expose functions and libraries that can produce random numbers which are predictable. These are not suited for cryptographic use! Make sure that you're using random libraries that are generating cryptographically strong random numbers. Usually the name of the library helps (for example you can probably guess which you should using between the `math/rand` and `crypto/rand` packages in Golang), but nothing replaces reading the manual!**

The following example shows how to generate some random bytes using PHP 7. These random bytes can be used by any cryptographic algorithm, for example as a secret key to encrypt with an authenticated-encryption algorithm. Every programming language does things differently, so make sure to consult your programming language's documentation in order to find out the standard way to obtain random numbers for cryptographic purposes.

Listing 8.2 random_numbers.php

```
<?php
$bad_random_number = rand(0, 10);   ❶

$secret_key = random_bytes(16);   ❷
?>
```

❶ This will produce a random number between 0 and 10. While fast, `rand` does not produce cryptographically secure random numbers! It is thus not suitable for cryptographic algorithms and protocols.

❷ `random_bytes` creates and fills a buffer with 16 random bytes. The result is suitable to be used for cryptographic algorithms and protocols.

Now that you've learned how you can obtain cryptographically secure randomness in your programs, let's think about the security considerations you need to keep in mind when you generate randomness.

8.4 Randomness generation and security considerations

It is good to remember at this point that any useful protocol based on cryptography requires good randomness, and that a broken PRNG could lead to the entire cryptographic protocol or algorithm being broken. It should be clear to you that a MAC is only as secure as the key used with it, or that the slightest ounce of predictability usually destroys signature schemes like ECDSA, and so on…

So far, this chapter makes it sound like generating randomness should be a simple part of applied cryptography, but in practice it is not. Randomness has actually been the source of many many bugs in real-world cryptography, due to a multitude of issues: using a non-cryptographic PRNG, badly seeding a PRNG (for example, using the current time which is predictable), and so on.

One example includes programs using **userland PRNGs**, as opposed to the kernel PRNGs which are behind system calls. Userland PRNGs usually add friction to a place where they are not needed, and can in the worst of cases cause devastating bugs if misused. This was notably the case with the PRNG offered by the OpenSSL library that was patched in some OS in 2006, inadvertently causing all SSL and SSH keys generated using the vulnerable PRNG to be affected.

> *Removing this code has the side effect of crippling the seeding process for the OpenSSL PRNG. Instead of mixing in random data for the initial seed, the only "random" value that was used was the current process ID. On the Linux platform, the default maximum process ID is 32,768, resulting in a very small number of seed values being used for all PRNG operations.*
>
> *– H. D. Moore Debian OpenSSL Predictable PRNG Toys (2008)*

For this reason, and others I will mention later in this chapter, it is wise to avoid userland PRNG and to stick to randomness provided by the OS when available. In most situations, sticking to what the programming language's standard library, or what a good cryptography library provides, should be enough.

> *We cannot keep on adding 'best practice' after 'best practice' to what developers need to keep in the back of their heads when writing everyday code.*
>
> *– Martin Boßlet OpenSSL PRNG Is Not (Really) Fork-safe (2013)*

Unfortunately, no list of advice can really prepare you for the many pitfalls of acquiring good randomness. Since randomness is at the center of every cryptography algorithm, making tiny mistakes can lead to devastating outcomes. It is good to keep in mind the following edge cases in case you run into them.

Forking processes. When using a userland PRNG (some applications with extremely high performance requirements might have no other choice), it is important to keep in mind that a program which forks will produce a new child process that will have the same PRNG state as its

parent. Consequently, both PRNGs will produce the same sequence of random numbers from there on. For this reason, if you really want to use a userland PRNG you have to be careful to make forks use different seeds for their PRNGs.

Virtual machines (VMs). This cloning of PRNG state can also become a problem when using the OS PRNG! Think about VMs. If the entire state of a VM is saved and then started several times from this point on, every instance might produce the exact same sequence of random numbers. This is sometimes fixed by hypervisors and OS's, but it is good to look into what the hypervisor you're using is doing before running applications that request random numbers in VMs.

Early boot entropy. While OS's should have no trouble gathering entropy in user-operated devices due to the noise produced by the user'sinteractions with the device, embedded devices and headless systems have more challenges to overcome in order to produce good entropy at boot time. History has shown that some devices will tend to boot in a similar fashion and end up amassing the same initial "noise" from the system, leading to the same seed being used for their internal PRNG and the same series of random numbers being generated.

> *There is a window of vulnerability —a boot-time entropy hole— during which Linux's urandom may be entirely predictable, at least for single-core systems. […] When we disabled entropy sources that might be unavailable on a headless or embedded device, the Linux RNG produced the same predictable stream on every boot.*
>
> *– Heninger et al. Mining Your Ps and Qs: Detection of Widespread Weak Keys in Network Devices (2012)*

In these rare cases where you really (really) need to obtain random numbers early during boot, one can help the system by providing some initial entropy generated from another machine's well-seeded `getrandom` or `/dev/urandom`. Different OS's might provide this feature and you should consult their manual (as usual) if you find yourself in this situation.

If available, a TRNG provides an easy solution to the problem. For example, modern Intel CPUs embed a special hardware chip that extracts randomness from thermal noise. This randomness is available through an instruction called `RDRAND`.

NOTE **Interestingly, Intel's `RDRAND` has been quite controversial due to fear of backdoors. Most OS's that have integrated `RDRAND` as a source of entropy mix it with other sources of entropy in a way that is contributory. Contributory here means that one source of entropy cannot force the outcome of the randomness generation. Imagine for a minute that mixing different sources of entropy was done by simply XORing them together; can you see how this might fail to be contributory?**

Finally, let me mention that one solution of avoiding the randomness pitfalls is to use algorithms

that **rely less on randomness**.

For example, you've seen in chapter 7 on signatures that ECDSA requires you to generate a random nonce every time you sign, whereas EdDSA does not. Another example you've seen in chapter 4 on authenticated encryption is AES-GCM-SIV, which does not catastrophically break down if you happen to reuse the same nonce twice, as opposed to AES-GCM which will leak the authentication key and will then lose integrity of the ciphertexts.

8.5 Public randomness

So far, I've talked mostly about "private" randomness, the kind you might need for your private keys. Sometimes, privacy is not required, and "public" randomness is needed. In this section I briefly survey some ways to obtain such public randomness. I will distinguish two cases:

- **One to many**. In this scenario you want to produce randomness for others.
- **Many to many**. In this scenario a set of participants want to produce randomness together.

First, let's imagine that you want to generate a stream of randomness in a way that many participants can verify it. In other words, the stream should be unpredictable, but impossible to alter from your perspective. Now imagine that you have a signature scheme which provides unique signatures based on a keypair and a message. With such a signature scheme, there exist a construction called a **verifiable random function (VRF)** to obtain random numbers in a verifiable way. It works like this:

- You generate a keypair and publish the verifying key. You also publish a public seed.
- To generate random numbers, you sign the public seed and hash the signature. The digest is your random number, the signature is also published as proof.
- To verify the random number, anyone can hash the signature to check if it matches the random number, and verify that the signature is correct with the public seed and verifying key.

I illustrate this concept in figure 8.6.

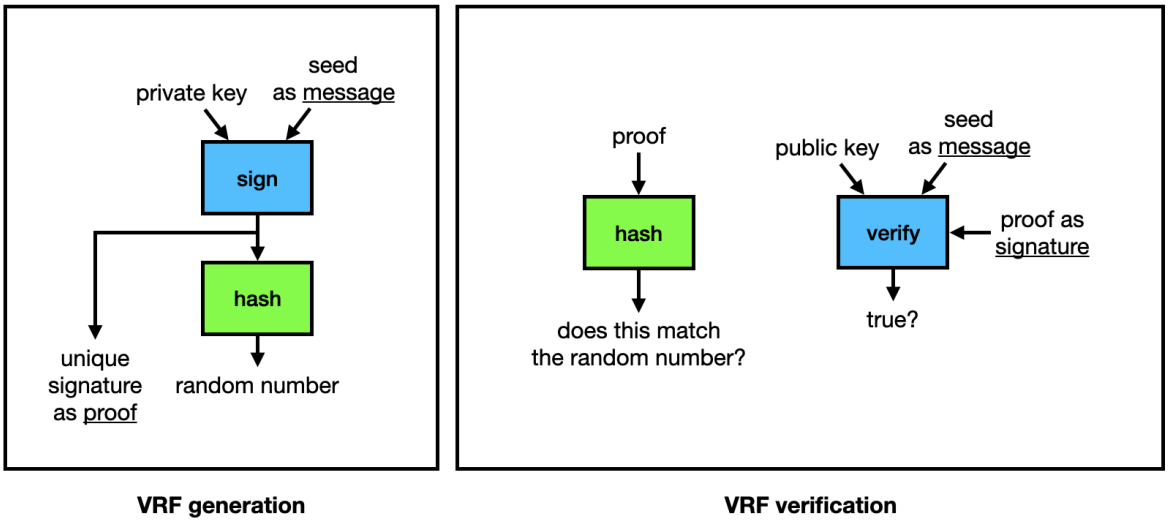


Figure 8.6 A Verifiable Random Function (VRF) generates verifiable randomness via public-keycrypto. To generate a radom number, simply use a signature scheme that produces unique signatures (like BLS) to sign a seed, then hash the signature to produce the public random number. To verify that randomness, make sure that the hash of the signature is indeed the random number, and verify the signature over the seed.

This construction can be extended to produce many random numbers by using the public seed like a counter. Since the signature is unique and the public seed is fixed, there is no way for the signer to generate a different random number.

By the way, signature schemes like BLS (mentioned in chapter 7) produce unique signatures, but this not true for ECDSA and EdDSA. (Do you see why?) To solve this, the Internet Draft (soon-to-be RFC) https://tools.ietf.org/html/draft-irtf-cfrg-vrf-08 specifies how to implement a VRF using ECDSA.

In some scenarios, for example a lottery game, several participants might want to randomly decide on a winner. We call these **decentralized randomness beacons**, as their role is to produce the same verifiable randomness even if some participants decides not to take part in the protocol. A common solution is to use the previously-discussed VRFs not with a single key but with a **threshold distributed key**: a key that is split between many participants and that will produce a unique valid signature for a given message only after a threshold of participants have signed the message. This might sound a bit confusing, as this is the first time I'm talking about distributed keys, know that you will learn more about them later in this chapter.

One popular decentralized randomness beacon is called **drand**, and is run in concert by several organizations and universities. It is available at https://leagueofentropy.com.

The main challenge in generating good randomness is that no party involved in the randomness generation process should be able to predict or bias the final output. A drand network is not controlled by anyone of its members. There is no single point of failure, and none of the drand server operators can bias the randomness generated by the network.

– https://drand.love How drand works (2021)

Now that I've talked extensively about randomness and how programs obtain it nowadays, let's move the discussion towards the role of secrets in cryptography and how one can manage them.

8.6 Key derivation with HKDF

PRNGs are not the only constructions one can use to derive more secrets from one secret (in other words, to "stretch" a key). Deriving several secrets from one secret is actually such a frequent pattern in cryptography that this concept was given its own name: **Key Derivation**. So let's see what this is about.

A **Key Derivation Function (KDF)** is like a PRNG in many ways, except for a number of subtleties:

- A KDF does not necessarily expect a uniformly random secret (as long as it has enough entropy). This makes a KDF very useful for deriving secrets from key exchanges output which produce high-entropy but biased results (see chapter 5 on key exchanges). The resulting secrets are in turn uniformly random and can be used in constructions that expect uniformly random keys.
- A KDF is usually used in protocols that require participants to rederive the same keys several times. In this sense, a KDF is expected to be deterministic, while PRNGs sometimes provide backward secrecy by frequently reseeding themselves with more entropy.
- A KDF is usually not designed to produce a LOT of random numbers, instead it is usually used to derive a limited number of keys.

These differences are summarized in figure 8.7.

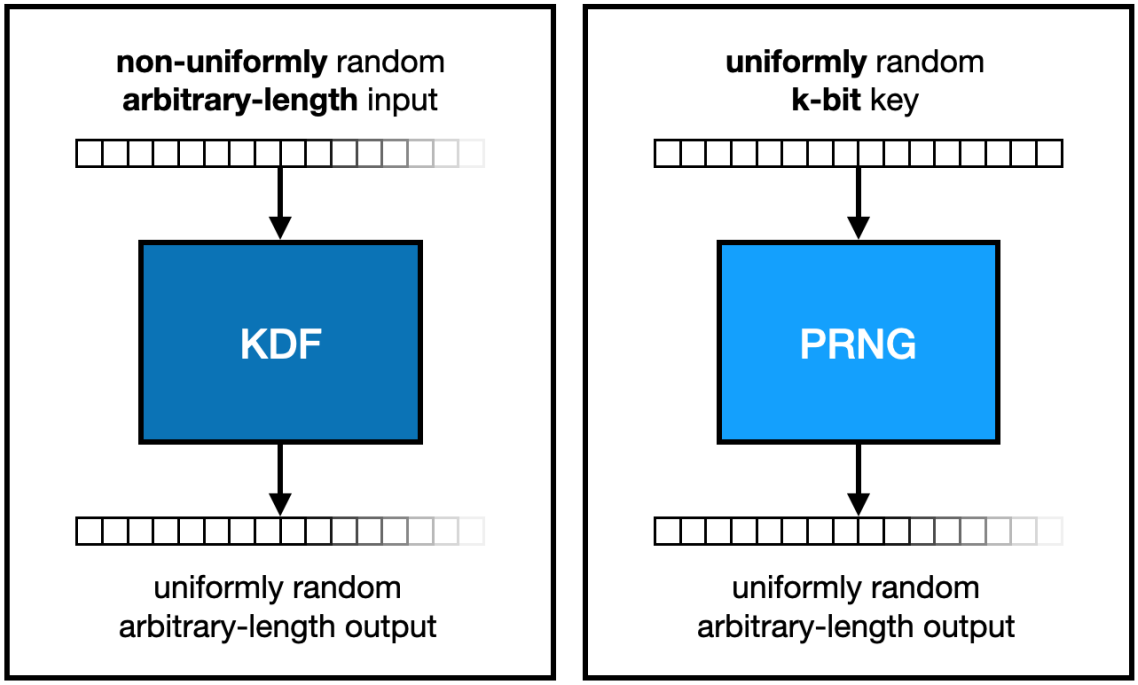


Figure 8.7 A Key Derivation Function (KDF) and a PRNG are two similar constructions. The main differences are that a KDF does not expect a fully uniformly random secret as input (as long as it has enough entropy), and is usually not used to generate too much output.

The most popular KDF is **HMAC-based Key Derivation Function (HKDF)**. You've learned about HMAC in chapter 3, a MAC based on hash functions. **HKDF** is a light KDF built on top of HMAC and defined in RFC 5869. For this reason, one can use HKDF with different hash functions (although it is most commonly used with SHA-2).

HKDF is specified as two different functions:

- **HKDF-Extract**. This function is used to remove biases from the input secret.
- **HKDF-Expand**. This function is more similar to a PRNG, it expects a uniformly random secret (and is thus run with the output of HKDF-Extract) and produces an arbitrary-length and uniformly random output.

Let's see HKDF-Extract first (which I illustrate in figure 8.8). Technically a hash function is enough to uniformize the randomness of an input bytestring (remember, the output of a hash function is supposed to be indistinguishable from randomness), but HKDF goes further and accepts one additional input: a `salt`. As for password hashing, a `salt` is here to differentiate different usages of HKDF-Extract in the same protocol. While this `salt` is optional (and set to an all-zero bytestring if not used), it is recommended to use it. Furthermore, HKDF does not expect the `salt` to be a secret; it can be known to everyone, including adversaries.

Instead of a hash function HKDF-Extract uses a MAC (specifically **HMAC**) which coincidentally has an interface that accepts two arguments.

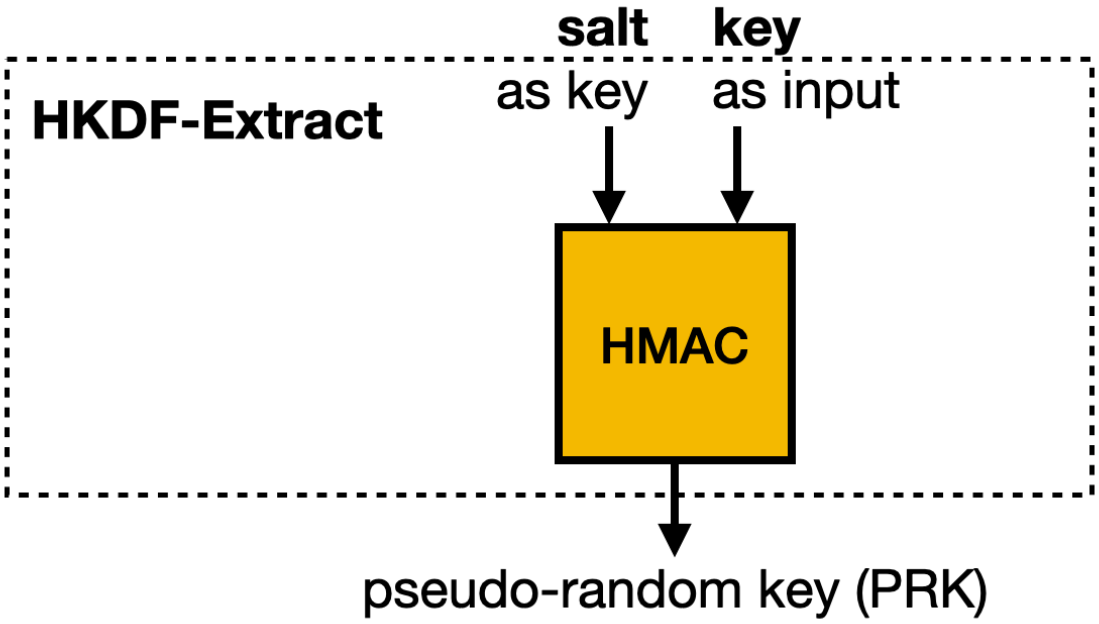


Figure 8.8 HKDF-Extract is the first function specified by HKDF. It takes an optional `salt` (which will be used as the key in HMAC) and the input secret that might be non-uniformly random. Using different `salt`s with the same input secret will produce a different output.

Let's now look at HKDF-Expand (which I illustrate in figure 8.9 below). If your input secret is already uniformly random, you can skip HKDF-Extract and directly use this one. Similar to HKDF-Extract, it also accepts an additional and optional customization argument called `info`. While salt is meant to provide some domain separation between calls of HKDF (or HKDF-Extract) within the same protocol, `info` is meant to be used to differentiate your version of HKDF (or HKDF-Expand) from other protocols. You can also specify how much output you want, but keep in mind that HKDF is not a PRNG and is not designed to derive a large number of secrets. HKDF is limited by the size of the hash function used, more precisely if you use SHA-512 (which produces outputs of 512 bits) with HKDF, you will be limited to 512 × 255 bits = 16,320 bytes of output for a given key and `info` bytestring.

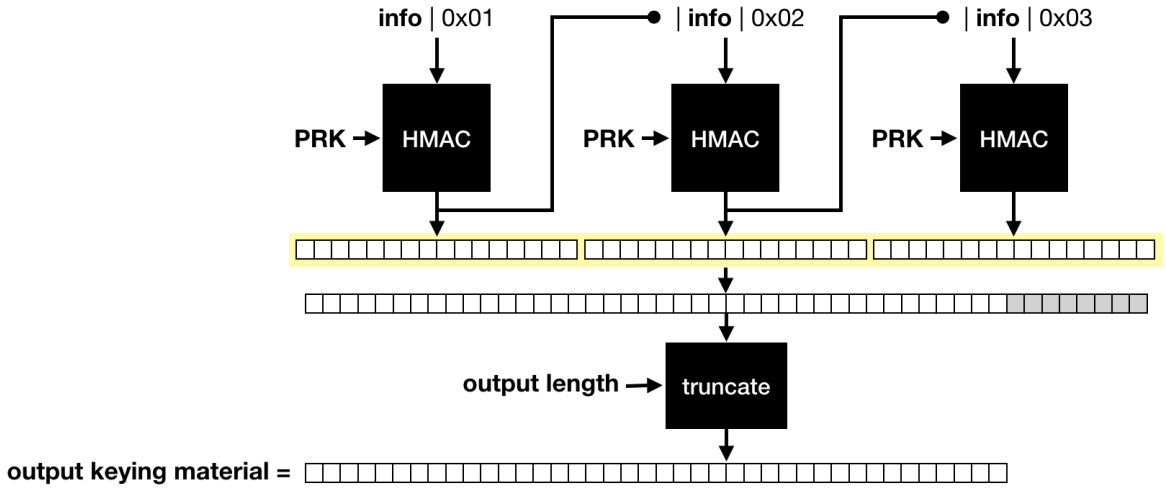


Figure 8.9 HKDF-Expand is the second function specified by HKDF. It takes an optional `info` bytestring and an input secret that needs to be uniformly random. using different `info` bytestrings with the same input secret will produce different outputs. The length of the output is controlled by a `length` argument.

Calling HKDF or HKDF-Expand several times with the same arguments except for the output length, will produce the same output truncated to the different length requested (see figure 8.10). This property is called **related outputs** and can, in rare scenarios, surprise protocol designers. It is good to keep this in mind.

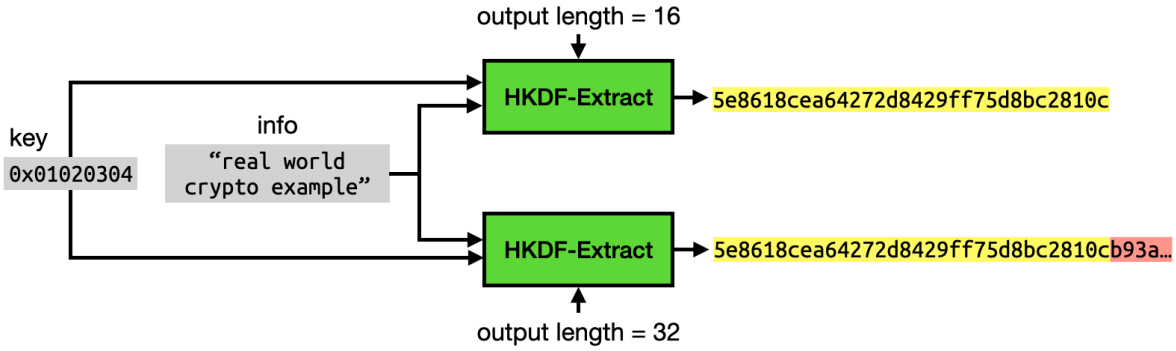


Figure 8.10 HKDF and HKDF Expands provide related outputs, meaning that calling the function with different output lengths will just truncate the same result to the requested length.

Most cryptographic libraries will combine the HKDF-Extract and HKDF-Expand into a single call, as illustrated in figure 8.11. As usual, make sure to read the manual (in this case RFC 5869) before using HKDF.

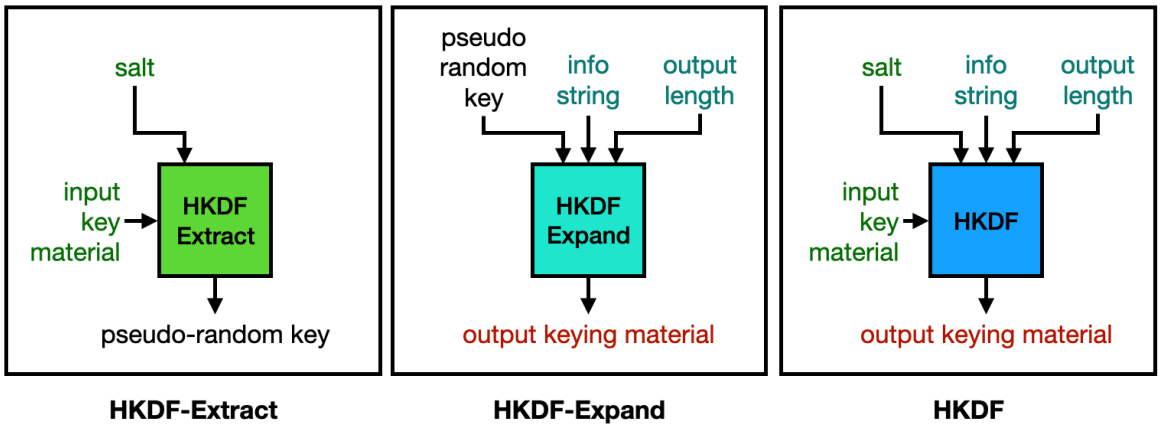


Figure 8.11 HKDF is usually implemented as a single call, that combines both HKDF-Extract (to extract uniform randomness from an input key) and HKDF-Expand (to generate an arbitrary-length output).

HKDF is not the only way to derive multiple secrets from one secret. A more naive approach is to use hash functions. As hash functions do not expect a uniformly random input, and produce uniformly random outputs, they are fit for the task. Hash functions are not perfect, though, as their interface does not take into account domain separation (no customization string argument) and their output length is fixed. Best practice is to avoid hash functions when you can use a KDF instead. Nonetheless, some well-accepted algorithms do use hash functions for this purpose. For example, you've learned in the previous chapter 7 about the Ed25519 signature scheme that hashes a 256-bit key with SHA-512 to produce two 256-bit keys.

NOTE **In theory, a hash function's properties do not say anything about the output being uniformly random, the properties only dictate that a hash function should be collision resistant, pre-image resistant and second pre-image resistant. In the real-world, though, we use hash functions all over the place to implement random oracles (as you've learned in chapter 2) and thus we assume that their outputs are uniformly random. This is the same with MACs which are in theory not expected to produce uniformly random outputs (unlike PRFs, as seen in chapter 3), but in practice do for the most part, and this is why HMAC is used in HKDF. In the rest of this book, I will assume that popular hash functions (like SHA-2 and SHA-3) and popular MACs (like HMAC and KMAC) produce random outputs.**

The extended output functions (XOFs) we've seen in chapter 2 (SHAKE and cSHAKE) can be used as a KDF as well! Remember, a XOF:

- does not expect a uniformly random input
- can produce a practically infinitely large uniformly random output

In addition KMAC (a MAC covered in chapter 3) does not have the related output issue I've mentioned earlier. Indeed, KMAC's length argument randomizes the output of the algorithm

(effectively acting like an additional customization string).

Finally, there exists an edge case for inputs that have low entropy. Think about passwords for example, that can be relatively guessable compared to a 128-bit key. The password-based key derivation functions used to "hash" passwords (covered in chapter 2) can also be used to derive keys as well.

8.7 Managing keys and secrets

Alright, all good, we know how to generate cryptographic random numbers and we know how to derive secrets in different types of situations. We're not out of the woods yet. Now that we're using all of these cryptographic algorithms, we end up having to maintain a lot of secret keys. How do we store these keys? And how do we prevent these extremely sensitive secrets from being compromised? And what do we do if a secret is compromised? This problem is commonly known as **key management**.

Crypto is a tool for turning a whole swathe of problems into key management problems.
– Lea Kissner in 2019

While many applications choose to leave keys close to the application that makes use of them, this does not necessarily mean that they have no recourse when bad things happen.

To prepare against an eventual breach, or a bug that would leak a key, most serious applications employ two defense-in-depth techniques:

- **Key rotation**. By associating an expiration date to a key (usually a public key), and by replacing your key with a new key periodically, you can heal from an eventual compromise. The shorter the expiration date and rotation frequency, the faster you will replace a key that might be known to an attacker.
- **Key revocation**. Key rotation is not always enough, and you might want to be able to "cancel" a key as soon as you hear it has been compromised. For this reason, some systems allow you to ask if a key has been revoked before making use of it. You will learn more about this in the next chapter 9 on secure transport.

Automation is often key to successfully using these techniques, as a well-oiled machine is much more apt to work correctly in times of crisis.

Furthermore, you can also associate a particular role to a key in order to limit the consequences of a compromise. For example, you could differentiate two public keys in some fabricated application as:

- public key 0x5678… is only for signing transactions
- public key 0x8530… is only for doing key exchanges

This would allow a compromise of the private key associated to public key 0x8530 not to impact transaction signing.

If one does not want to leave keys lying around on device storage media, hardware solutions exist that aim at preventing keys from being extracted. You will learn more about them in chapter 13 on hardware cryptography.

Finally, there exist a lot of ways for applications to delegate key management. This is often the case on mobile OS's that provide "key stores" or "key chains" which will keep keys for you and even perform cryptographic operations. Applications living in the cloud can sometimes have access to cloud key management services. These services allow an application to delegate creation of secret keys and cryptographic operations and avoid thinking about the many ways to attack them.

Nonetheless, as with hardware solutions, if an application is compromised it will still be able to do any type of request to the delegated service. There are no silver bullets and you should still consider what you can do to detect and respond to a compromise.

Key management is a hard problem that is beyond the scope of this book, so I will not dwell on this topic too much.

In the next section, I will go over cryptographic techniques that attempts at avoiding the key management problem!

8.8 Avoiding key management, or how to decentralize trust

Key management is a vast field of study, which can be quite annoying to invest in as users do not always have the resources to implement best practices or tools available in the space.

Fortunately, cryptography has something to offer for those who wish to lessen the burden of key management.

The first one I'll talk about is **secret sharing** (or **secret splitting**). Secret splitting allows you to break a secret in multiple parts that can be shared among a set of participants.

Here a secret can be anything you want, a symmetric key, a signing private key, and so on.

Typically, a person called a dealer will generate the secret, then split it and share the different parts among all participants before deleting the secret. I illustrate this process in figure 8.12>.

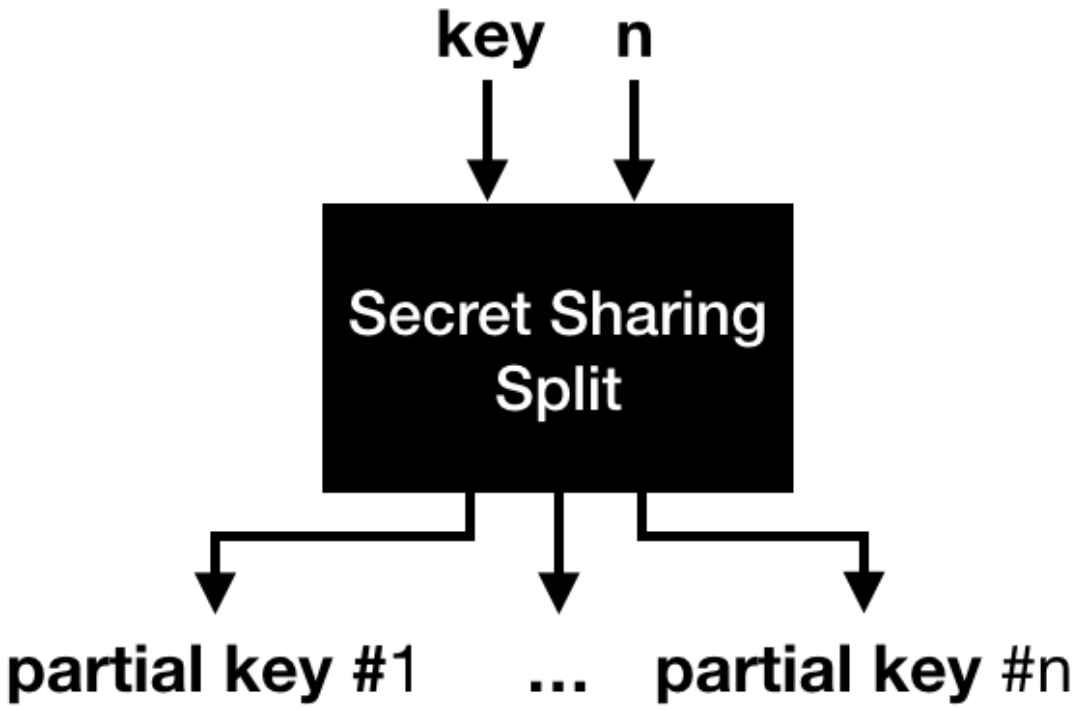


Figure 8.12 Given a key and a number of shares n, the Shamir's Secret Sharing scheme creates n partial keys of the same size as the original key.

When time comes, and the secret is needed to perform some cryptographic operation (encrypting, signing, and so on), all share owners need to provide their private shares back to the dealer who will be in charge of reconstructing the original secret.

Such a scheme prevents attackers from targeting a single user, as each share is useless by itself, and instead forces attackers to compromise all the participants before they can exploit a key. I illustrate this in figure 8.13.

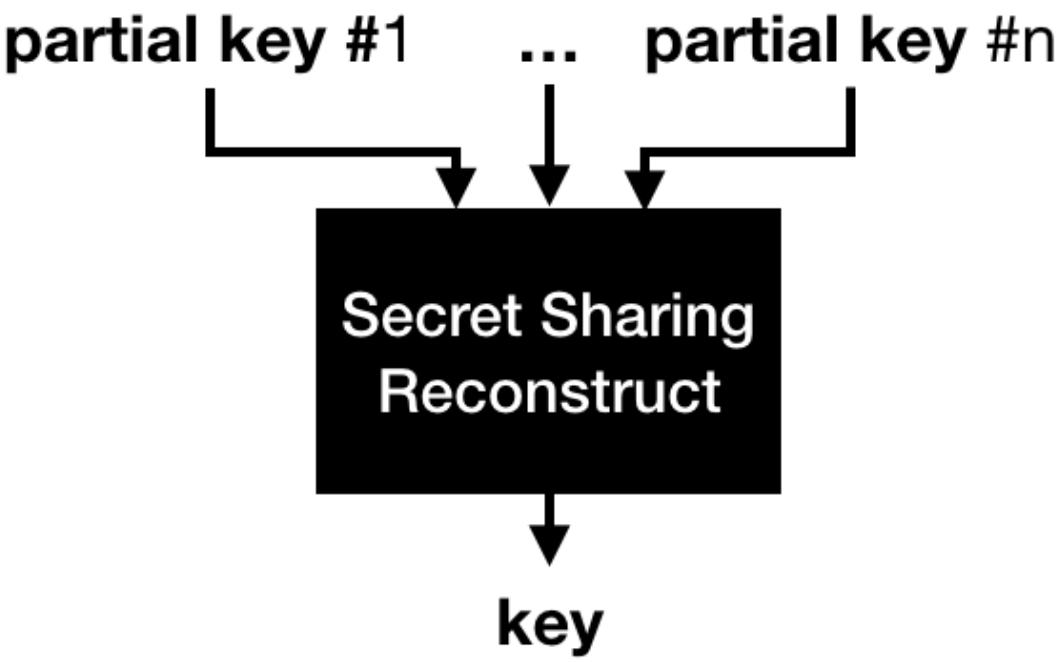


Figure 8.13 The Shamir's Secret Sharing scheme used to split a secret in n partial keys, will require all of the n partial keys to reconstruct the original key.

The most famous secret splitting scheme was invented by Adi Shamir (one of the co-inventor of RSA) and is called **Shamir's Secret Sharing (SSS)**.

The mathematics behind the algorithm are actually not too hard to understand! So let me spare a few paragraphs here to give you a simplified idea of the scheme.

Imagine a random straight line on a 2-dimensional space, and let's say that its equation $y = ax + b$ is the secret.

By having two participants hold two random points on the line, they can collaborate to recover the line equation. This is illustrated in figure 8.14.

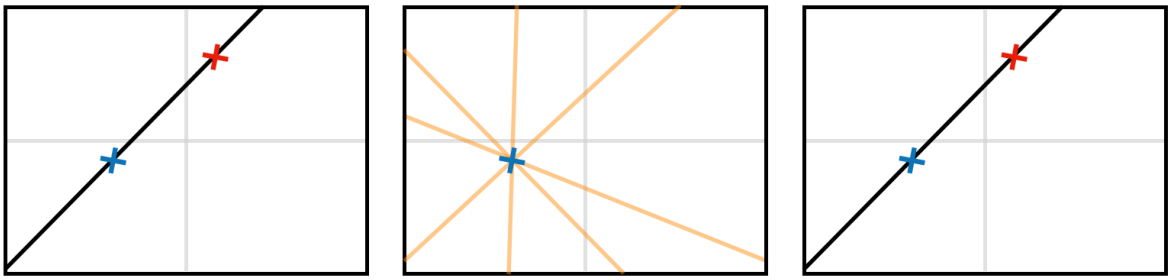


Figure 8.14 The idea behind the Shamir's Secret Sharing scheme is to see a polynomial defining a curve as the secret, and random points on the curve as partial keys. To recover a polynomial of degree n defining a curve, one needs to know n+1 points on the curve. For example, $f(x) = 3x + 5$ is of degree 1 so you need any two points $(x, f(x))$ to recover the polynomial; $f(x) = 5x^2 + 2x + 3$ is of degree 2 so you need any three points to recover the polynomial.

The scheme generalizes to polynomials of any degree, and thus can be used to divide a secret into an arbitrary number of shares.

Secret splitting is a technique often adopted due to its simplicity. Yet, in order to be useful key shares must be gathered into one place in order to recreate the key, every time the key has to be used in a cryptographic operation. This creates a window of opportunity in which the secret becomes vulnerable to robberies or accidental leaks, effectively getting us back to a **single point of failure** model.

To avoid this single point of failure issue, there exist several cryptographic techniques that can be useful in different scenarios.

For example, imagine a protocol that accepts a financial transaction only if it has been signed by Alice. This places a large burden on Alice, who might be scared of getting targeted by attackers. In order to reduce the impact of an attack on Alice, we can change the protocol to accept a number n of signatures instead (on the same transaction) from n different public keys, including Alice's. An attacker would thus have to compromise all n signatures in order to forge a valid transaction. Such systems are called **multi-signature** (often shortened as multi-sig) and have seen a lot of adoption in the cryptocurrency space.

Naive multi-signature schemes can add some annoying overhead. Indeed, the size of a transaction in our example grows linearly with the number of signatures required. To solve this, some signature schemes (like the **BLS signature scheme**) can have their signatures compressed down to a single signature. This is called **signature aggregation**.

Some multi-signature schemes go even further in the compression by allowing the n public keys to be aggregated into a single public key. This technique is called **distributed key generation (DKG)** and is part of a field of cryptography called **secure multi-party computation** (which I

will cover in chapter 15). DKG lets n participants collaboratively compute a public key without ever having the associated private key in the clear during the process (unlike SSS, there is no dealer). If they want to sign a message, they can then collaboratively create a signature (using each participant's private shares) that can be verified using the public key they previously created. Again, the private key never exists physically, preventing the single point of failure problem SSS had. Since you've seen Schnorr signatures in chapter 7, figure 8.15 shows the intuition behind a simplified schnorr DKG scheme.

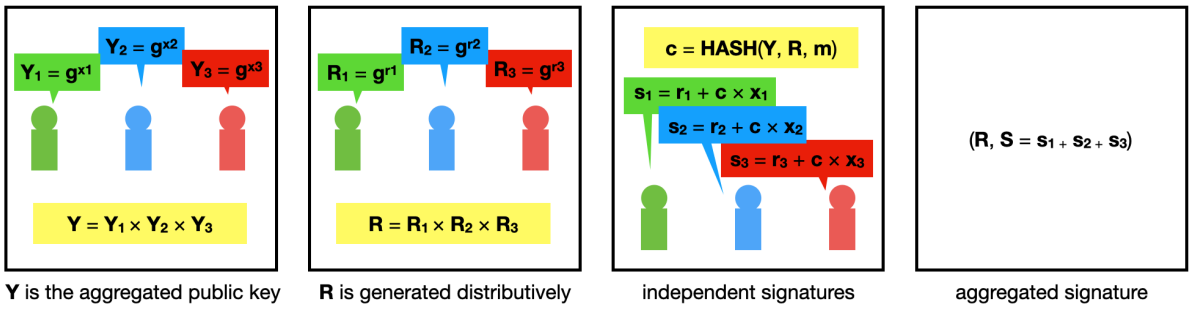


Figure 8.15 The Schnorr signature scheme can be decentralized into a distributed key generation scheme.

Finally, note that:

- These types of schemes can work with other asymmetric cryptographic algorithms like encryption, where participants must all collaborate to asymmetrically decrypt a message, for example.
- Each scheme I've mentioned can be made to work even when only a threshold *m* out of the *n* participants helps. Although the term **threshold signatures** often refers to the threshold version of DKG.

I recapitulate all these examples in figure 8.16.

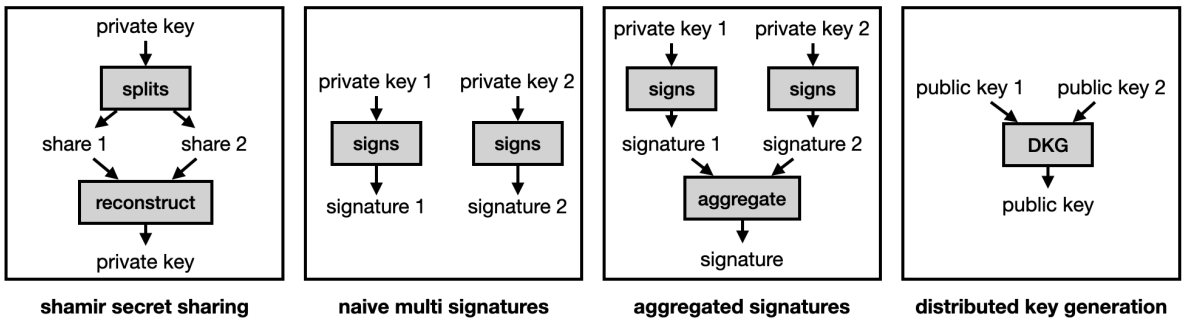


Figure 8.16 A recap of existing techniques to split trust we have in one participant into several participants.

8.9 Summary

- A number is taken uniformly and at random from a set, if it was picked with equal probability compared to all other numbers from that set.
- Entropy is a metric to indicate how much randomness a bytestring has. High-entropy refers to bytestrings that are uniformly random, while low-entropy refers to bytestrings that are easy to guess or predict.
- Pseudo-Random Number Generators (PRNGs) are algorithms that take a uniformly random seed, and generate (in practice an infinite) amount of randomness that can be used for cryptographic purposes (as cryptographic keys, for example). That is if the seed is large enough.
- To obtain random numbers, one should rely on a programming language's standard library or on its well-known cryptographic libraries.
- If these are not available, OS's generally provide interfaces to obtain random numbers: Windows offers the `BCryptGenRandom` system call; Linux and FreeBSD offer the `getrandom` system call; Other OS's usually have a special file `/dev/urandom` exhibiting randomness.
- Key Derivation Functions (KDFs) are useful in scenarios where one wants to derive secrets from a biased but high-entropy secret.
- HKDF is the most widely used KDF, and is based on HMAC.
- Key management is the field of keeping secrets, well, secret. It mostly consists of finding where to store secrets, proactively expiring and rotating secrets, figuring out what to do when secrets are compromised.
- To lessen the burden of key management, one can split the trust of one participant into multiple participants.

9 Secure transport

This chapter covers

- Secure transport protocols, protocols used to encrypt communications between machines.
- The Transport Layer Security (TLS) protocol, the most widely used secure transport protocol.
- The Noise protocol framework, a modern alternative to TLS.

The heaviest use of cryptography today is most probably to encrypt communications. After all, cryptography was invented for this purpose. To do this, applications generally do not make use of cryptographic primitives (like authenticated encryption) directly, but instead use much more involved protocols that abstract the use of the cryptographic primitives. I call these protocols "secure transport" protocols, for lack of a better term.

In this chapter you will learn about the most widely-used secure transport protocol: the Transport Layer Security (TLS) protocol. I will also lightly cover other secure transport protocols and how they differ from TLS.

9.1 The SSL and TLS secure transport protocols

In order to understand why transport protocols are a thing, let's walk through a motivating scenario.

When you enter http://example.com in your web browser, your browser uses a number of protocols to connect to a web server, and to retrieve the page you requested. One of them is the **Hypertext Transfer Protocol (HTTP)**, which your browser use to tell the web server on the other side which page it is interested in. HTTP uses a human-readable format. This means that

you can look at the HTTP messages that are being sent and received over the wire, and read them without the help of any other tool.

But this is not enough for your browser to communicate to the web server. HTTP messages are encapsulated into another type of messages called TCP frames, which are defined in the Transmission Control Protocol (TCP). TCP is a binary protocol, and thus it is not human-readable: you need a tool to understand the fields of a TCP frame.

TCP messages are furhter encapsulated using the Internet Protocol (IP), and IP messages are further encapsulated using something else. This is known at the internet protocol suite, and as it is the subject of many books,[1] I won't go much further into this.

Back to our scenario; there's a confidentiality issue that we need to talk about. Anyone sitting on the wire in between your browser and the web server of example.com has a very interesting position: they can passively observe and read your requests as well as the server's responses. Worse, the MITM attacker can also actively tamper and reorder messages.

This is not great. Imagine your credit card information leaking every time you buy something on the internet, your passwords being stolen when you log into a website, your pictures and private messages stolen as you send them to your friends.

This scared enough people that in the 90s the successor of TLS, the **Secure Sockets Layer (SSL)** protocol, was born. While SSL could be used in different kinds of situations, it was first built by and for web browsers.[2] As such, it started being used in combination with HTTP, extending it into the **Hypertext Transfer Protocol Secure (HTTPS)**. HTTPS now allowed browsers to secure their communications to the different websites they visited.

9.1.1 From SSL to TLS

While SSL was not the only protocol that attempted to secure some of the web, it did attract most of the attention and with time has become the de-facto standard. But this is not the whole story. Between the first version of SSL and what we currently use today, a lot has happened! All versions of SSL (the last being SSL version 3.0) were broken due to a combination of bad design and bad cryptographic algorithms. [3]

After SSL 3.0, the protocol was officially transferred to the Internet Engineering Task Force (IETF), the organization in charge of publishing **Request For Comments (RFCs)** standards. The name SSL was dropped in favor of **Transport Layer Security (TLS)**, and TLS 1.0 was released in 1999 as RFC 2246. The most recent version of TLS is **TLS version 1.3**, specified in RFC 8446 and published in 2018. TLS 1.3, unlike its predecessor, stems from a solid collaboration between the industry and academia. Yet, today, the internet is still divided between many different versions of SSL and TLS, as servers have been slow to update.

> **NOTE** There's a lot of confusion around the two names SSL and TLS. The protocol is now of course called TLS, but many articles, and even libraries, still choose to use the term SSL.

TLS has become more than just the protocol securing the web, it is now being used in many different scenarios and between many different types of applications and devices as a protocol to secure communications. Thus what you will learn about TLS in this chapter is not just useful for the web, but for any scenario where communications between two applications need to be secured.

9.1.2 Using TLS in practice

So how do people use TLS? First let's define some terms. In TLS, the two participants that want to secure their communications are called a **client** and a **server**. It works the same way as with other network protocols like TCP or IP: the client is the one that initiates the connection.

A **TLS client** is typically built from:

- Some **configuration**. A client is configured with the versions of SSL and TLS that it wants to support, cryptographic algorithms that it is willing to use to secure the connection, ways it can authenticate servers, and so on.
- Some **information about the server it wants to connect to**. It includes at least an IP address and a port, but for the web it often includes a fully qualified domain name instead (like example.com).

Given these two arguments, a client can initiate a connection with a server to produce a secure **session**: a channel that both the client and the server can use to share encrypted messages to each other. In some cases a secure session cannot successfully be created and fails midway. For example, if an attacker attempts to tamper with the connection, or if the server's configuration is not compatible with the client's (more on that later), the client will fail to establish a secure session.

A **TLS server** is often much simpler as it only take a **configuration**, which is very similar to the configuration of the client. A server then waits for clients to connect to it, in order to produce a secure session.

In practice, using TLS on the client side can be as easy as the following snippet of code, that is if you use a programming language like Golang:

Listing 9.1 tls_server.go

```
import "crypto/tls"

func main() {
    destination := "google.com:443"   ❶
    TLSconfig := &tls.Config{}   ❷
    conn, err := tls.Dial("tcp", destination, TLSconfig)
    if err != nil {
        panic("failed to connect: " + err.Error())
    }
    conn.Close()
}
```

❶ The fully qualified domain name and the port of the server (443 is the default port for HTTPS).

❷ An empty config serves as default configuration.

How does the client know that the connection it established is really with google.com and not some impersonator? By default Golang's TLS implementation uses your operating system configuration to figure out how to authenticate TLS servers. Later in this chapter, you will learn exactly how the authentication in TLS works.

Using TLS on the server side is pretty easy as well, as the following code snippet shows:

Listing 9.2 tls_server.go

```
import "crypto/tls"

func main() {
    config := &tls.Config{   ❶
        MinVersion: tls.VersionTLS13,   ❶
    }   ❶

    http.HandleFunc("/", func(rw http.ResponseWriter,
       req *http.Request) {   ❷
        rw.Write([]byte("Hello, world\n"))   ❷
    })   ❷

    server := &http.Server{   ❸
        Addr:      ":8080",   ❸
        TLSConfig: config,   ❸
    }

    err := server.ListenAndServeTLS("cert.pem", "key.pem")   ❹
    if err != nil {
        log.Fatal(err)
    }
}
```

❶ A solid minimal configuration for a TLS 1.3 server.

❷ Serves a simple page displaying "Hello, world."

❸ An HTTPS server is started on port 8080.

❹ Some `.pem` files containing a certificate and a secret key (more on this later).

Golang and its standard library do a lot for us here. Unfortunately not all languages' standard libraries provide easy-to-use TLS implementations (if they provide a TLS implementation at all), and not all TLS libraries provide secure-by-default implementations. For this reason, configuring a TLS server is not always straightforward depending on the library.

> **NOTE** Note that TLS is a protocol that works on top of TCP. To secure UDP connections, DTLS (D is for datagram, the term for UDP messages) can be used and is fairly similar to TLS, for this reason I will ignore DTLS in this chapter.

In the next section, you will learn about the inner workings of TLS and its different subtleties.

9.2 How does the TLS protocol work?

As I said earlier, today TLS is the de-facto standard to secure communications between applications. In this section you will learn more about how TLS works underneath the surface, and how it is used in practice. You will find this section useful for understanding how to use TLS properly, and also for understanding how most (if not all) secure transport protocols work. (You will also find out why it is hard and strongly discouraged to redesign or reimplement such protocols.)

At a high level, TLS is split into two phases:

- A **handshake** phase where a secure communication is negotiated and created between two participants.
- A **post-handshake** phase where communications are encrypted between the two participants.

This idea is shown in illustration 9.1

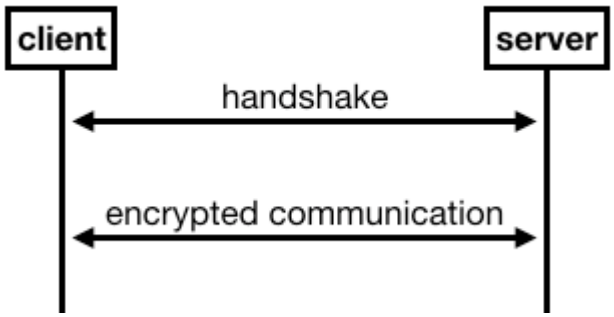


Figure 9.1 At a high level, secure transport protocols first create a secure connection during a handshake phase. After that, applications on both sides of the secure connection can communicate securely.

At this point, since you have learned about hybrid encryption in chapter 6, you should have the

following (correct) intuition about how these two steps works:

- The handshake is, at its core, simply a key exchange. The handshake ends up with the two participants agreeing on a set of symmetric keys.
- The post-handshake phase is purely about encrypting messages between participants, using an authenticated encryption algorithm and the set of keys produced at the end of the handshake.

Most transport security protocols work this way, and the interesting parts of these protocols always lie in the handshake phase.

Next, let's take a look at the handshake phase.

9.2.1 The TLS handshake

As you've seen, TLS is (and most transport security protocols are) divided into two parts: a handshake and a post-handshake phase. In this section you'll learn about the handshake first. The handshake itself has 4 aspects that I want to tell you about:

1. Negotiation. TLS is highly configurable. Both a client and a server can be configured to negotiate a range of SSL and TLS versions, as well as a menu of acceptable cryptographic algorithms. The negotiation phase of the handshake aims at finding common ground between the client's and the server's configurations, in order to securely connect the two peers.
2. Key exchange. The whole point of the handshake is to perform a key exchange between the two participants. What key exchange algorithm to use? This is one of the things decided as part of the negotiation process.
3. Authentication. As you've learned in chapter 5 on key exchanges, it is trivial for a MITM attacker to impersonate any side of a key exchange. Due to this, key exchanges must be authenticated. (Your browser must have a way to make sure that it is talking to google.com and not your internet service provider, for example.)
4. Session Resumption. As browsers often connect to the same websites again and again, key exchanges can be costly and slow down a user's experience. For this reason, mechanisms to fast-track secure sessions without redoing a key exchange are integrated into TLS.

This is a long list. As fast as greased lightning, let's start with the first item.

NEGOTIATION IN TLS: WHAT VERSION AND WHAT ALGORITHMS?

Most of the complexity in TLS comes from the negotiation of the different moving parts of the protocol. Infamously, this negotiation has also been the source of many issues in the history of TLS. Attacks like FREAK, LOGJAM, DROWN, and others… took advantage of weaknesses present in older versions to break more recent versions of the protocol (as long as a server offered to negotiate these versions). While not all protocols have versioning, or allow for different algorithms to be negotiated, SSL/TLS was designed for the web. As such, SSL/TLS needed a way to maintain backward compatibility with older clients and servers that could be slow to update.

This is what happens on the web today: your browser might be recent and up-to-date, made to support TLS 1.3, but visiting some old web page chances are that the server behind it only supports versions of TLS up to 1.2 or 1.1 (or worse). Vice-versa, many websites must support older browsers (and thus older versions of TLS) as some users are stuck in the past.

> **NOTE** **Most versions of SSL and TLS have security issues, except for TLS version 1.2 and 1.3. So why not just support the latest version 1.3 and call it a day? Some companies support older clients that can't easily be updated: these might be hardware devices or businesses that are slow to update. Due to these requirements, it is not uncommon to find libraries implementing mitigations to known attacks in order to support older versions securely. Unfortuantely, these mitigations are often too complex to implement right. For example, well-known attacks like Lucky13 and Bleichenbacher98 have been rediscovered again and again by security researchers in various TLS implementations that had previously attempted to fix the issues. So while it is possible to mitigate a number of attacks on older versions of TLS, I would recommend against.**

Negotiation starts with the client sending a first request called a **Client Hello** to the server. The Client Hello contains a range of supported SSL and TLS versions, a suite of cryptographic algorithms that the client is willing to use, and some more information that can be relevant for the rest of the handshake or for the application. The suite of cryptographic algorithms include:

- One or more **key exchange** algorithms. TLS 1.3 defines the following algorithms allowed for negotiations: ECDH with P-256, P-384, P-521, X25519, X448; and FFDH with the groups defined in RFC 7919. I've talked about all of these in chapter 5. Previous versions of TLS also offered RSA key exchanges (covered in chapter 6) but they were removed in the last version.
- Two (for different parts of the handshake) or more **digital signature** algorithms. TLS 1.3 specifies RSA PKCS#1 v1.5 and the newer RSA-PSS, as well as more recent elliptic curve algorithms like ECDSA and EdDSA. I've talked about these in chapter 7. Note that digital signatures are specified with a hash function which allows you to negotiate, for example, RSA-PSS with either SHA-256 or SHA-512.

- One or more **hash functions** to be used with HMAC (the message authentication code you've learned in chapter 3) and HKDF (the key derivation function covered in chapter 8). TLS 1.3 specifies SHA-256 and SHA-384, two instances of the SHA-2 hash function. You've learned about SHA-2 in chapter 2. This choice of hash function is unrelated to the one used by the digital signature algorithm.
- One or more **authenticated encryption** algorithms. These can include AES-GCM (with keys of 128 or 256 bits), ChaCha20-Poly1305, and AES-CCM. I've talked about all of these in chapter 4.

The server then responds with a **Server Hello** message that contains one of each type of cryptographic algorithms, cherry-picked from the client's selection.

If the server is unable to find an algorithm it supports, it has to abort the connection. Although in some cases, the server does not have to abort the connection and can ask the client to provide more information instead. To do this, the server replies with a message called a Hello Retry Request asking for specific piece of information. The client can then resend its Client Hello, this time with the added requested information.

TLS AND FORWARD SECURE KEY EXCHANGES

The key exchange is the most important part of the TLS handshake! Without it, there's obviously no symmetric key being negotiated. But for a key exchange to happen, the client and the server must first trade their respective public keys.

In TLS 1.2 and previous versions, the client and the server starts a key exchange only after both participants agree on which key exchange algorithm to use (during a negotiation phase).

TLS 1.3 optimizes this flow by attempting to do both the negotiation and the key exchange at the same time: the client speculatively chooses a key exchange algorithm and sends a public key in the very first message (the Client Hello). If the client fails to predict the server's choice of key exchange algorithm, then the client falls back to the outcome of the negotiation, and sends a new Client Hello containing the correct public key. For example:

- The client sends a TLS 1.3 Client Hello announcing that it can do either an X25519 or an X448 key exchange. It also sends an X25519 public key.
- The server does not support X25519, but does support X448. It sends a Hello Retry Request to the client announcing that it only supports X448.
- The client sends the same Client Hello but with an X448 public key instead.
- The handshake goes on.

I illustrate this difference in figure 9.2.

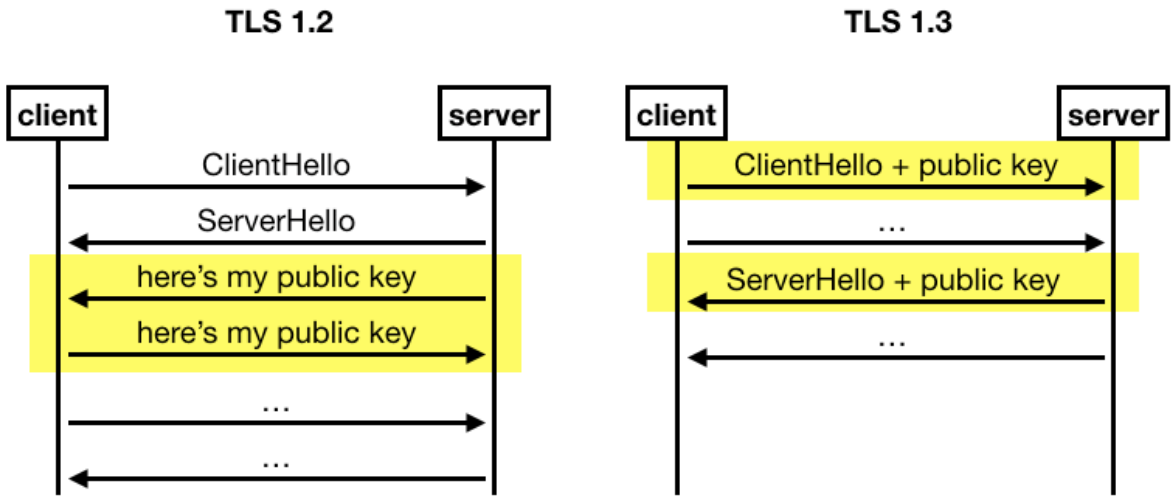


Figure 9.2 In TLS 1.2, the client waits for the server to choose which key exchange algorithm to use before sending a public key. In TLS 1.3, the client speculates on which key exchange algorithm(s) the server will settle on, and preemptively sends a public key (or several) in the first message, potentially avoiding an extra round trip.

TLS 1.3 is full of such optimizations, which are important for the web. Indeed many people in the world have unstable or slow connections, and it is important to keep non-application communication to the bare minimum required.

Furthermore, in TLS 1.3 and unlike previous versions of TLS, all key exchanges are **ephemeral**. This means that for each new session, the client and the server both generate new key pairs, then get rid of them as soon as the key exchange is done. This provides **forward secrecy** to the key exchange: a compromise of the long-term keys of the client or the server won't allow an attacker to decrypt this session. That is, as long as the ephemeral private keys were safely deleted.

Imagine what would happen if instead, a TLS server used a single private key for every key exchange it performed with its clients.

A compromise of the server's private key at some point in time would be devastating, as a MITM attacker would then be able to decrypt all previously recorded conversations. (Do you understand how?)

Instead, by performing ephemeral key exchanges and getting rid of private keys as soon as a handshake ends, the server protects against such attackers. I illustrate this in figure 9.3.

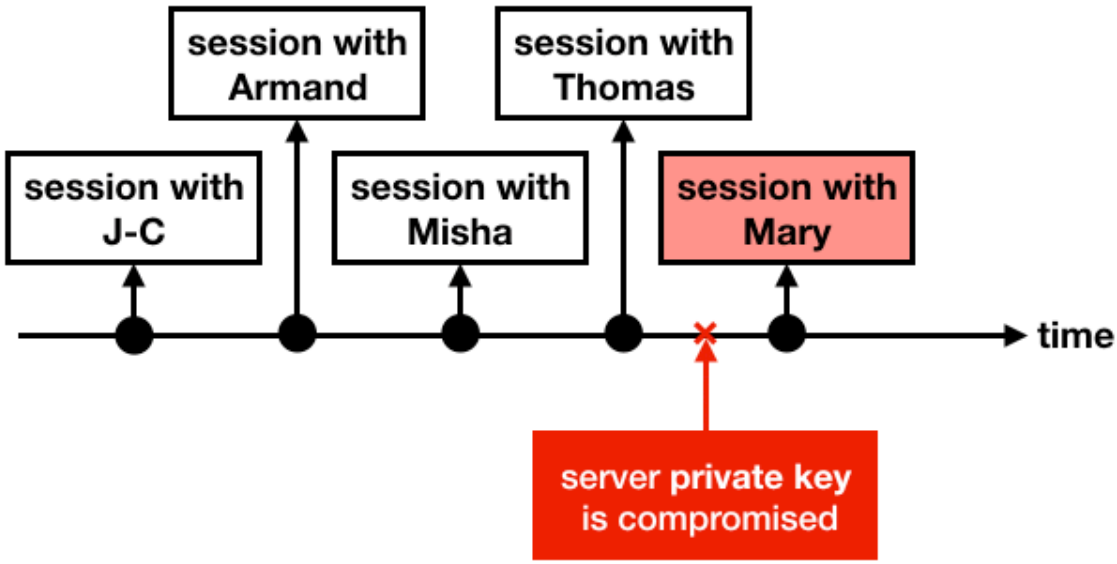


Figure 9.3 In TLS 1.3 each session starts with an ephemeral key exchange. If a server is compromised at some point in time, no previous sessions will be impacted.

Once ephemeral public keys are traded, a key exchange is performed, and keys can be derived. TLS 1.3 derives different keys at different points in time, to encrypt different phases with independent keys.

The first two messages, the Client Hello and the Server Hello, cannot be encrypted as no public keys were traded at this point. But after that, as soon as the key exchange happens, TLS 1.3 encrypts the rest of the handshake. (This is unlike previous versions of TLS that did not encrypt any of the handshake messages.)

To derive the different keys, TLS 1.3 uses HKDF with the hash function negotiated. HKDF-Extract is used on the output of the key exchange (to remove any biases) while HKDF-Expand is used with different `info` parameters to derive the encryption keys. For example, "c hs traffic" (for client handshake traffic) is used to derive symmetric keys for the client to encrypt to the server during the handshake, "s ap traffic" (for "server application traffic") is used to derive symmetric keys for the server to encrypt to the client after the handshake.

Remember, unauthenticated key exchanges are insecure; next you'll see how TLS addresses this.

TLS AUTHENTICATION AND THE WEB PUBLIC KEY INFRASTRUCTURE

After some negotiations, and after the key exchange has taken place, the handshake must go on. What happens next is the other most important part of TLS: **authentication**. You've seen in chapter 5 on key exchanges that it is trivial to intercept a key exchange and impersonate one (or both) sides of the key exchange. In this section I'll explain how your browser cryptographically validates that it is talking to the right website, and not to an impersonator.

But let's take a step back. There is something I haven't told you so far. A TLS 1.3 handshake is actually split in three different stages (as illustrated in figure 9.4):

1. **Key Exchange**. This phase contains the ClientHello and ServerHello messages which provide some negotiation and perform the key exchange. All messages (including handshake messages) after this phase are encrypted.
2. **Server Parameters**. Messages in this phase contain additional negotiation data from the server. This is negotiation data that does not have to be contained in the first message of the server, and that could benefit from being encrypted.
3. **Authentication**. This phase includes authentication information from both the server and the client.

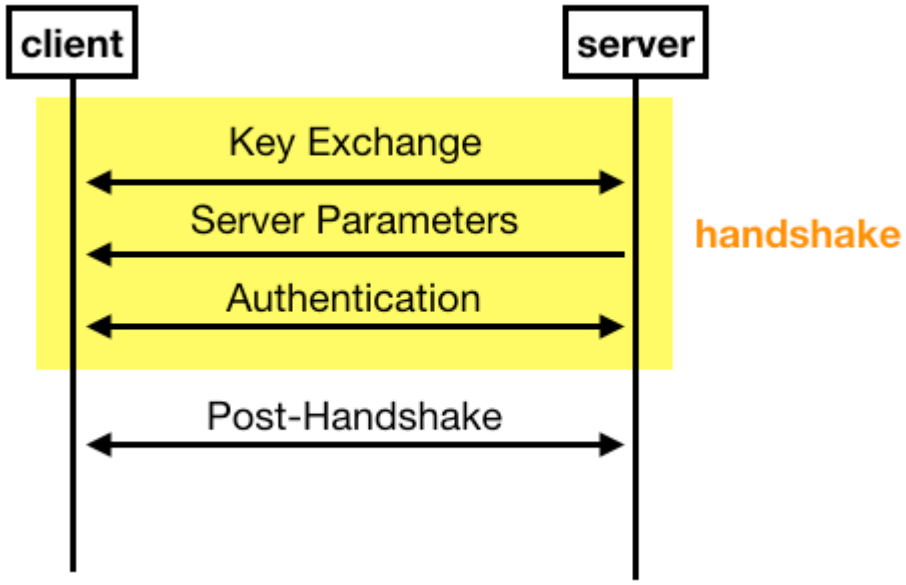


Figure 9.4 A TLS 1.3 handshake is divided into 3 phases: the key exchange phase, the server parameters phase, and finally the authentication phase.

On the web, **authentication in TLS is usually one-sided**. Only the browser verifies that google.com is indeed google.com, but google.com does not verify who you are (or at least not as part of TLS).

NOTE Client authentication is often delegated to the application layer for the web (via a form asking you for your credentials most often). That being said, client authentication can also happen in TLS if requested by the server (during the server parameters phase). When both sides of the connection are authenticated, we talk about mutually-authenticated TLS (sometimes abbreviated as mTLS). Client authentication is done the exact same way as server authentication, and can happen at any point in time after the authentication of the server (during the handshake or in the post-handshake phase).

Let's now answer the question: when connecting to google.com, how does your browser verify that you are indeed handshaking with google.com?

Using the **web public key infrastructure (web PKI)**!

You've learned about the concept of public key infrastructure in chapter 7 on digital signatures, but let me briefly reintroduce this concept as it is quite important to understand how the web works.

There are two sides to the web PKI:

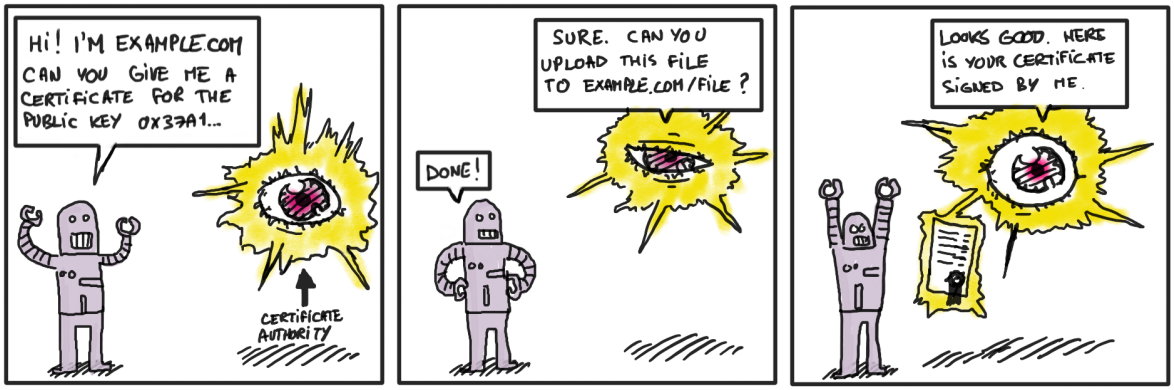
First, **Browsers must trust a set of root public keys** that we call **Certificate Authorities (CAs)** . Usually, browsers will either use a hardcoded set of trusted public keys or rely on the operating system to provide them.

NOTE For the web, there exist hundreds of these CAs which are independently run by different companies and organizations across the world. It is quite a complex system to analyze and these CAs can sometimes also sign the public keys of intermediate CAs that in turn also have the authority to sign the public keys of websites. For this reason, organizations like the Certification Authority Browser Forum (CA/Browser forum) enforce rules and decide when new organizations can join the set of trusted public keys, and when a CA can no longer be trusted and must be removed from that set.

Second, **websites who want to use HTTPS must have a way to obtain a certification from these CAs** (a signature of their signing public key). In order to do this, a website owner (or a webmaster as we used to say) must prove to a CA that they own a specific domain.

NOTE Obtaining a certificate for your own website used to involve a fee, this is no longer the case nowadays as Certificate Authorities like Let's Encrypt provide these for free.

For example, to prove that you own example.com a CA might ask you to host a file at example.com/some_path/file.txt that contains some random numbers generated for your request.



After this, a CA can provide a signature over the website's public key. As the CA's signature is usually valid for a period of years, we say that it is over a **long-term** signing public key (as opposed to an ephemeral public key). More specifically, CAs do not actually sign public keys, but instead they sign **certificates** (more on this later). A certificate contains the long-term public key, along with some additional important metadata like the name of your domain if you are a web page.

To prove to your browser that the server it is talking to is indeed google.com, the server sends a **certificate chain** as part of the TLS handshake which comprises of:

1. Its own (leaf) certificate containing among others the name google.com, google's long-term signing public key, as well as a signature from a CA.
2. A chain of intermediate CA certificates, from the one that signed google's certificate to the root CA that signed the last intermediate CA.

This is a bit wordy so I illustrated this in figure 9.5.

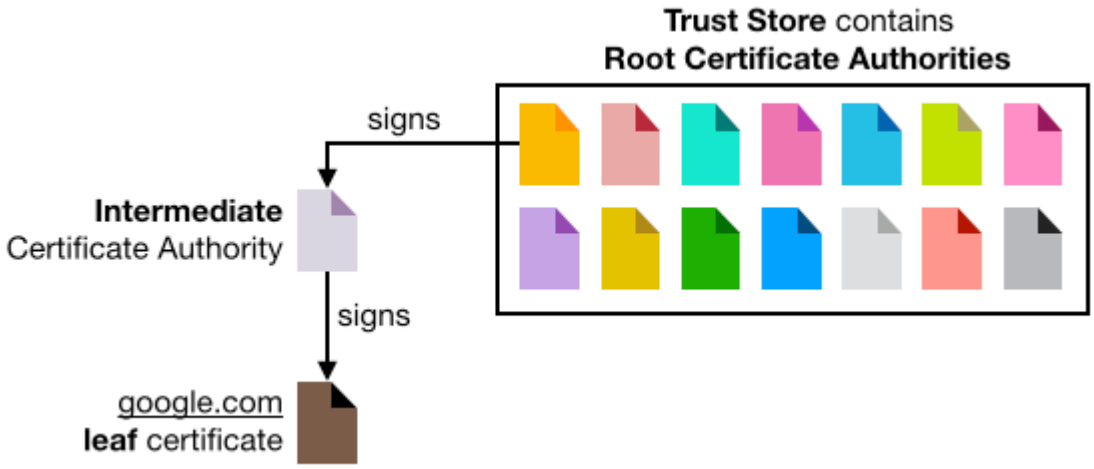


Figure 9.5 Web browsers only have to trust a relatively small set of root Certificate Authorities (CAs) in order to trust the whole web. These CAs are stored in what is called a trust store. In order for a website to be trusted by a browser, the website must have its (leaf) certificate be signed by one of these CAs. Sometimes root CAs only sign intermediate CAs, who in turn sign other intermediate CAs or leaf certificates. This is called the web public key infrastructure (web PKI).

This certificate chain is sent in a **Certificate** TLS message by the server, and by the client as well if the client has been asked to authenticate.

Following this, the browser can use its certified long-term key pair to sign all handshake messages that have been received and sent since then (in what is called a **Certificate Verify** message).

I've recapitulated this flow (where only the server authenticates itself) in figure 9.6.

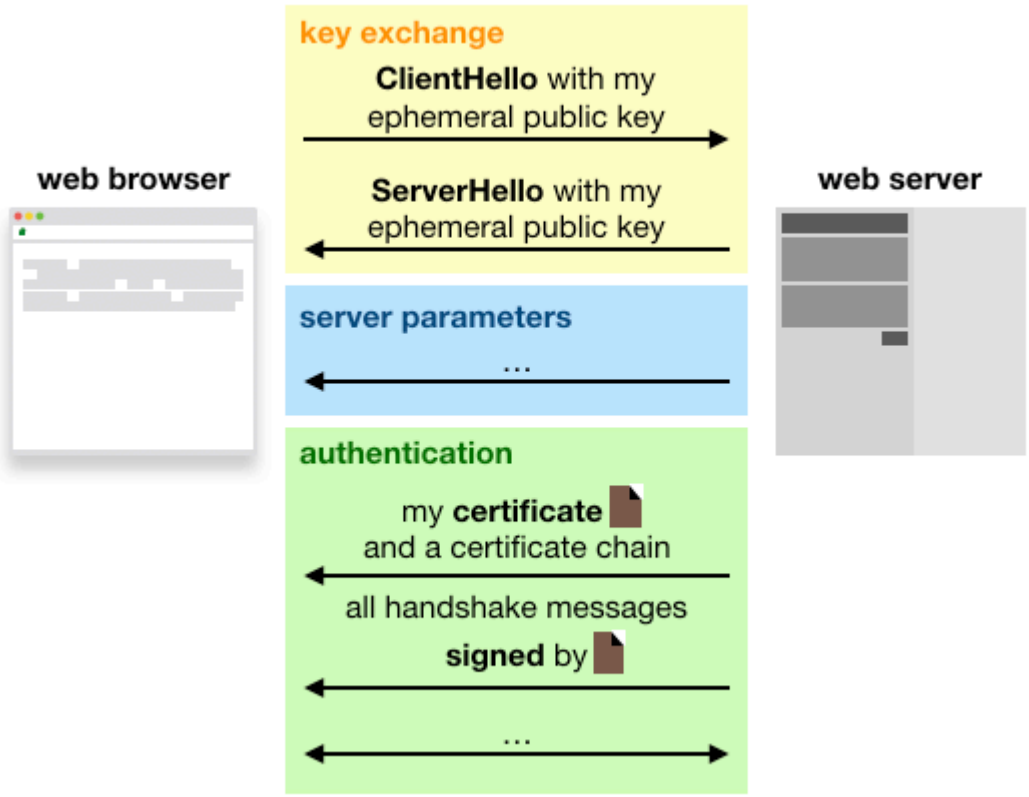


Figure 9.6 The authentication part of a handshake starts with the server sending a certificate chain to the client. The certificate chain starts with the leaf certificate (the certificate containing the website's public key and additional metadata like the domain name) and ends with a root certificate that is trusted by the browser. Each certificate contains a signature from the certificate above it in the chain.

The signature in the Certificate Verify message proves to the client what the server has seen so far. Without this signature, a MITM attacker could intercept the server's handshake messages and replace the ephemeral public key of the server contained in the Server Hello message, allowing the attacker to successfully impersonate the server.

Take a few moments to understand why an attacker cannot replace the server's ephemeral public key in the presence of the Certificate Verify signature.

SIDEBAR **Story time.**

A few years ago I was hired to review a custom-TLS protocol made by a large company. It turned out that their protocol had the server provide a signature that did not cover the ephemeral key. When I told them about the issue, the whole room went silent for a full minute.

It was of course a substantial mistake: an attacker who could have intercepted the custom handshake and replaced the ephemeral key with its own, would have successfully impersonated the server. The lesson here is that it is important not to reinvent the wheel. Secure transport protocols are hard to get right and if history has shown anything, they can fail in many unexpected ways. Instead, you should rely on mature protocols like TLS and make sure you use a popular implementation that has received a substantial amount of public attention.

Finally, in order to officially end the handshake, both sides of the connection must send a **Finished** message as part of the Authentication phase. A Finished message contains an authentication tag produced by HMAC (instantiated with the negotiated hash function for the session). This allows both the client and the server to tell the other side: "these are all the messages I have sent and received, in order, during this handshake". If the handshake was intercepted and tampered with by a MITM attacker, this integrity check can allow the participants to detect and abort the connection. (This is especially useful as some handshakes modes are not signed, more on this later).

Before heading to a different aspect of the handshake, let's double click on X.509 certificates, as they are an important detail of many cryptographic protocols.

AUTHENTICATION VIA X.509 CERTIFICATES

While certificates are optional in TLS 1.3 (you can always use plain keys), many applications and protocols (not just the web) make heavy use of them in order to certify additional metadata. Specifically, the **X.509 certificate standard version 3** is used.

X.509 is a pretty old standard that was meant to be flexible enough to be used in a multitude of scenarios, from email to web pages. The X.509 standards uses a description language called **Abstract Syntax Notation One (ASN.1)** to specify information contained in a certificate. An ASN.1 looks like this:

```
Certificate  ::=  SEQUENCE  {
    tbsCertificate       TBSCertificate,
    signatureAlgorithm   AlgorithmIdentifier,
    signatureValue       BIT STRING  }
```

You can literally read this as a structure that contains three fields:

- `tbsCertificate`. The "to-be-signed" certificate. This contains all the information that one wants to certify. For the web, this can contain a domain name (google.com), a public key, an expiration date, and so on.
- `signatureAlgorithm`. The algorithm used to sign the certificate.
- `signatureValue`. The signature from a Certificate Authority.

By the way, the last two values are not contained in the actual certificate (`tbsCertificate`), do you know why?

You can easily check what's in an X.509 certificate by connecting to any website using HTTPS, and using your browser functionalities to observe the certificate chain sent by the server. See figure 9.7 for an example.

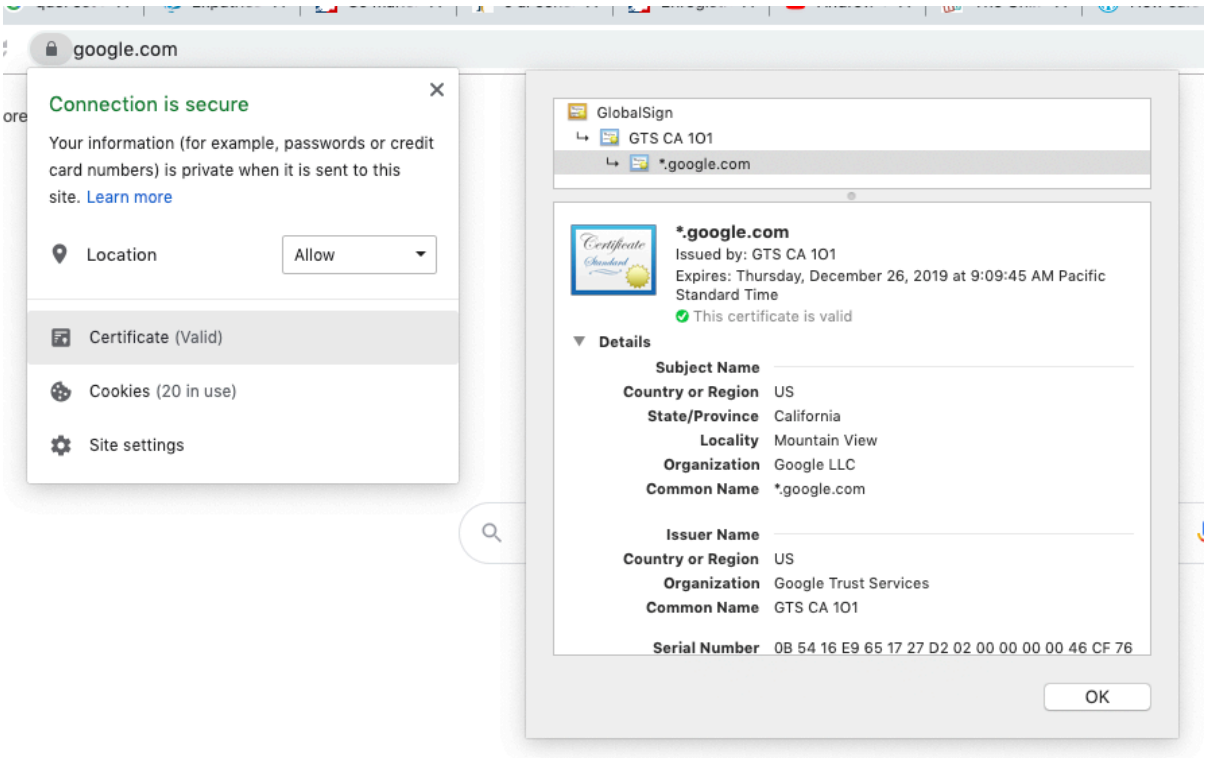


Figure 9.7 Using Chrome's Certificate viewer, we can observe the certificate chain sent by a Google's server. The root Certificate Authority is "Global Sign," which is trusted by your browser. Down the chain, an intermediate CA called "GTS CA 101" is trusted due to its certificate containing a signature from "Global Sign." In turn, Google's leaf certificate, valid for *.google.com (google.com, mail.google.com, and so on), contains a signature from "GTS CA 101."

You might encounter X.509 certificates as `.pem` files, which is some base64 encoded content surrounded by some human-readable hint of what the base64 encoded data contains (here a certificate). The following snippet represents the content of a certificate in a `.pem` format:

```
-----BEGIN CERTIFICATE-----
MIIJQzCCCCugAwIBAgIQC1QW6WUXJ9ICAAAAAEbPdjANBgkqhkiG9w0BAQsFADBC
MQswCQYDVQQGEwJVUzEeMBwGA1UEChMVR29vZ2xlIFRydXN0IFNlcnZpY2VzMRMw
EQYDVQQDEwpHVFMgQ0EgMU8xMB4XDTE5MTAwMzE3MDk0NVoXDTE5MTIyNjE3MDk0
NVowZjELMAkGA1UEBhMCVVMxEzARBgNVBAgTCkNhbGlmb3JuaWExFjAUBgNVBAcT
[...]
```

```
vaoUqelfNJJvQjJbMQbSQEp9y8EIi4BnWGZjU6Q+q/3VZ7ybR3cOzhnaLGmqiwFv
4PNBdnVVfVbQ9CxRiplKVzZSnUvypgBLryYnl6kquh1AJS5gnJhzogrz98IiXCQZ
c7mkvTKgCNIR9fedIus+LPHCSD7zUQTgRoOmcB+kwY7jrFqKn6thTjwPnfB5aVNK
dl0nq4fcF8PN+ppgNFbwC2JxX08L1wEFk2LvDOQgKqHR1TRJ0U3A2gkuMtf6Q6au
3KBzGW6l/vt3coyyDkQKDmT61tjwy5k=
-----END CERTIFICATE-----
```

If you decode the base64 content surrounded by the `BEGIN CERTIFICATE` and `END CERTIFICATE`, you end up with a **Distinguished Encoding Rules (DER)** encoded certificate. DER is a deterministic (only one way to encode) binary encoding used to translate X.509 certificates into bytes. All these encodings are often quite confusing to new-comers! I recap all of this in figure 9.8.

Figure 9.8 On the top left corner, an X.509 certificate is written using the ASN.1 notation. It is then transformed into bytes that can be signed via the DER encoding. As this is not text that can easily be copied around and be recognized by humans, it is base64 encoded. The last touch wraps the base64 data with some handy contextual information using the PEM format.

DER only encodes information as "here is an integer" or "this is a bytearray." Fields' names described in ASN.1 like `tbsCertificate` are thus lost after encoding. Decoding DER without the knowledge of the original ASN.1 description of what each field truly means is thus pointless. Handy command line tools like OpenSSL allow you to decode and translate in human terms the content of a DER-encoded certificate. For example, if you download google.com's certificate, you can use the following command to display its content in your terminal:

```
$ openssl x509 -in google.pem -text
Certificate:
    Data:
        Version: 3 (0x2)
        Serial Number:
            0b:54:16:e9:65:17:27:d2:02:00:00:00:00:46:cf:76
        Signature Algorithm: sha256WithRSAEncryption
        Issuer: C = US, O = Google Trust Services, CN = GTS CA 1O1
        Validity
            Not Before: Oct  3 17:09:45 2019 GMT
            Not After : Dec 26 17:09:45 2019 GMT
```

```
        Subject: C = US, ST = California, L = Mountain View,
        O = Google LLC, CN = *.google.com
        Subject Public Key Info:
            Public Key Algorithm: id-ecPublicKey
                Public-Key: (256 bit)
                pub:
                    04:74:25:79:7d:6f:77:e4:7e:af:fb:1a:eb:4d:41:
                    b5:27:10:4a:9e:b8:a2:8c:83:ee:d2:0f:12:7f:d1:
                    77:a7:0f:79:fe:4b:cb:b7:ed:c6:94:4a:b2:6d:40:
                    5c:31:68:18:b6:df:ba:35:e7:f3:7e:af:39:2d:5b:
                    43:2d:48:0a:54
                ASN1 OID: prime256v1
                NIST CURVE: P-256
[...]
```

Having said all of this, X.509 certificates are quite controversial.

Validating X.509 certificates has been comically dubbed "The Most Dangerous Code in the World" by a team of researchers in 2012.[4] This is because DER encoding is a difficult protocol to parse correctly, and the complexity of X.509 certificates makes for many mistakes to be potentially devastating. For this reason I don't recommend any modern application to use X.509 certificates, unless they have to.

PRE-SHARED KEYS AND SESSION RESUMPTION IN TLS, OR HOW TO AVOID KEY EXCHANGES

Key exchanges can be costly, and are sometimes not needed. For example, you might have two machines that only connect to each other, and you might not want to have to deal with a public key infrastructure in order to secure their communications. TLS 1.3 offers a way to avoid this overhead with **pre-shared keys (PSK)**.

A pre-shared key is simply a secret that both the client and the server know, and that can be used to derive symmetric keys for the session.

In TLS 1.3, a PSK handshake works by having the client advertise in its Client Hello message that it supports a list of PSK identifiers. If the server recognizes one of them, it can say so in its response (the Server Hello message) and both can avoid doing a key exchange (if they want to). By doing this, the authentication phase is skipped, making the Finished message at the end of the handshake important to prevent MITM attacks.

NOTE **Note that this does not mean that the same set of symmetric keys is derived for every session using the same PSK. Using different keys for different sessions is extremely important, as you do not want these sessions to be linked. Worse, since encrypted messages might be different between sessions, this could lead to nonce reuses and their catastrophic implications (see chapter 4). To mitigate this, both the Client Hello and Server Hello messages have a `random` field which is randomly generated for every new session. These random fields are used in the derivation of symmetric keys in TLS, effectively creating never-seen-before encryption keys every time you create a new connection.**

Another use case for PSKs is **session resumption**. Session resumption is about reusing secrets created from a previous session or connection. If you have already connected to google.com and have already verified their certificate chain and agreed on a shared secret, why do this dance again a few minutes or hours later? If you know how browsers work, you probably know that they also create several TCP connections to a web page they visit in order to quickly load a page; do we really want to do a full TLS handshake for all of these separate TCP connections as well?

TLS 1.3 offers a way to generate a PSK after a handshake was successfully performed, which can be used in subsequent connections to avoid having to redo a full handshake.

If the server wants to offer this feature, it can send **New Session Ticket** message(s) at any time during the post-handshake phase. The server can create so-called "session tickets" in several ways. For example, the server can send an identifier, associated to the relevant information in a database (forcing the server to keep a state), or the server can send the authenticated encryption of the required information to perform session resumption with the client (allowing the server's session resumption mechanism to be stateless). These are not the only ways, but as this mechanism is quite complex and most of the time not necessary, I won't touch more of it in this chapter.

Next let's see the easiest part of TLS: how communications eventually get encrypted.

9.2.2 How TLS 1.3 encrypts application data

Once a handshake has taken place, and symmetric keys have been derived, both the client and the server can send each other encrypted application data. This is not all: TLS also ensures that such messages cannot be replayed nor reordered.

To do this, the nonce used by the authenticated encryption algorithm starts at a fixed value and is incremented for each new message. If a message is replayed, or reordered, the nonce will be different from what is expected and decryption will fail. When this happens the connection is killed.

> **NOTE** As you've learned in chapter 4, encryption does not always hide the length of what is being encrypted. TLS 1.3 comes with record padding, which can be configured to pad application data with a random number of zero bytes before encrypting it, effectively hiding the true length of the message. In spite of this, statistical attacks that remove the added noise can exist, and it is not straightforward to mitigate them. If you really require this security property, you should refer to the TLS 1.3 specification.

Starting in TLS 1.3, if a server decides to allow it, clients have the possibility to send encrypted data as part of their first series of messages (right after the Client Hello message). This means that browsers do not necessarily have to wait until the end of the handshake to start sending application data to the server. This mechanism is called **early data** or **0-RTT** (for zero round-trip-time) and can only be used with the combination of a PSK (as it allows derivation of symmetric keys during the Client Hello message).

This feature was quite controversial during the development of the standard, as a passive attacker can replay an observed Client Hello followed by the encrypted 0-RTT data. This is why 0-RTT must be used only with application data that can be replayed safely.

For the web, browsers treat every GET queries as *idempotent*, meaning that they should not change any state on the server side and are only meant to retrieve data (unlike POST queries, for example). This is of course not always the case, and applications have been known to do whatever they want to do. For this reason, if you are confronted with the decision of using 0-RTT or not, it is simpler just not to use it.

9.3 The state of the encrypted web today

Today, standards are pushing for the deprecation of all versions of SSL and TLS that are not TLS 1.2 and TLS 1.3. Yet, due to legacy clients and servers, many libraries and applications continue to support older versions of the protocol (up to SSL version 3 sometimes!). This is not straightforward and due to the number of vulnerabilities you need to defend against, many hard-to-implement mitigations must be maintained.

> **WARNING** Using TLS 1.3 (and TLS 1.2) is considered secure and best practice. Using anything lower means that you will need to consult experts and will have to figure out how to avoid known vulnerabilities.

By default, browsers still connect to web servers using HTTP and websites still have to manually ask a CA to obtain a certificate. This means that with the current protocols, the web will never be fully encrypted (although some estimates global web traffic to be 90% encrypted in 2019).[5]

The fact that by default, your browser always uses an insecure connection is also an issue. Web servers nowadays usually redirect users accessing their page using HTTP towards HTTPS. Web servers can also (and often do) tell browsers to use HTTPS for subsequent connections. This is done via an HTTPS response header called **HTTP Strict Transport Security (HSTS)**. Yet, the very first connection to a website is still unprotected (unless the user thinks about typing "https" in the address bar) and can be intercepted to remove the redirection to HTTPS.

In addition, other web protocols like NTP (to get the current time) and DNS (to obtain the IP behind a domain name) are currently largely unencrypted and vulnerable to man-in-the-middle attacks. While there are research efforts to improve the status quo,[67] these are limitations one needs to be aware of today.

There's another threat to TLS users, and this is misbehaving CAs. What if today, a CA decides to sign a certificate for your domain and a public key that they control? If they can obtain a man-in-the-middle position, they could start impersonating your website to your users.

The obvious solution if you control the client-side of the connection is to either not use the web PKI (and rely on your own PKI), or to **pin** a specific certificate or public key. **Certificate or public key pinning** is simply about hardcoding in the client code the hash of the leaf certificate, or the public key that the leaf certificate contains, and to only accept a server that uses these for the authentication phase of the handshake. This practice is often used in mobile applications, as they know exactly what the server's public key or certificate should look like (unlike browsers that have to connect to an infinite number of servers).

This is not always possible though, and two other mechanisms exist:

- **Certificate revocation**. Like the name indicates, this allows a CA to revoke a certificate and warn browsers about it.
- **Certificate monitoring.** This is a relatively new system that forces CAs to publicly log every certificate they sign.

The story of certificate revocation has historically been bumpy. The first solution proposed was **Certificate Revocation Lists (CRLs)**, which allowed CAs to maintain a list of certificates that they had revoked (were no longer considered valid). The problem with CRLs is that they could grow quite large, and that one needs to constantly check them. CRLs were deprecated in favor of **Online Certificate Status Protocol (OCSP)**, which are simple web interfaces that one can query to know if a certificate is revoked or not. OCSP has its own share of problems: it requires CAs to have a highly-available service that can answer to OCSP requests, it leaks web traffic information to the CAs, and browsers often decide to ignore OCSP requests that time out (to not disrupt the user's experience).

The current solution is to augment OCSP with **OCSP Stapling**: the website is in charge of querying the CA for a signed status of its certificate, and attaches (staples) the response to its

certificate during the TLS handshake.

I recapitulate the three solutions in figure 9.9.

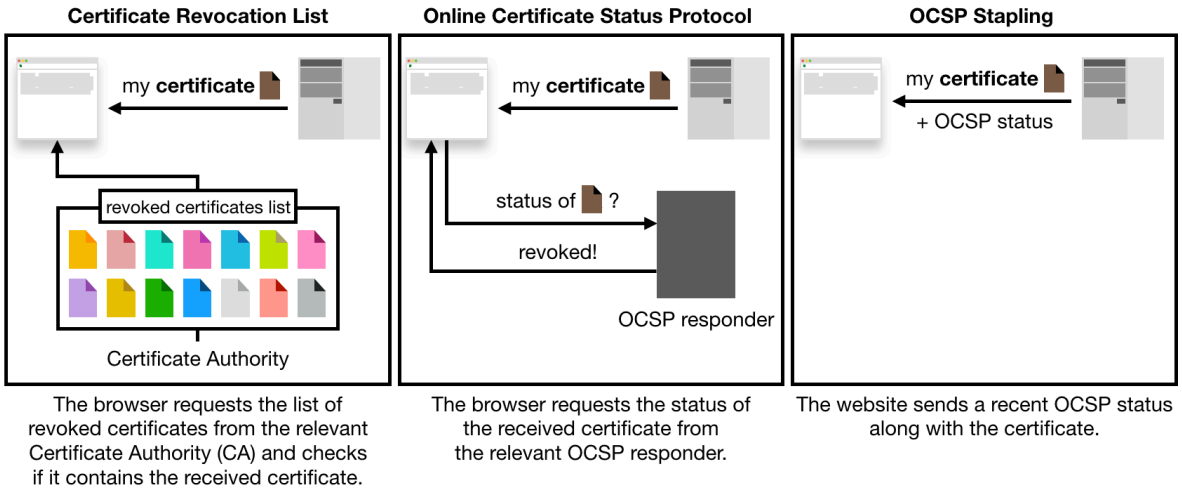


Figure 9.9 Certificate revocation on the web has had three popular solutions: Certificate Revocation Lists (CRL), Online Certificate Status Protocol (OCSP), and OCSP Stapling.

Certificate revocation might not seem to be a prime feature to support (especially for smaller systems compared to the web), that is until a certificate gets compromised. Like a car seatbelt, certificate revocation is a security feature that is useless most of the time, but can be a lifesaver in rare cases (what we call **defense-in-depth** in security).

> **NOTE** **For the web, certificate revocation has largely proven to be a good decision. In 2014, the Heartbleed bug turned out to be one of the most devastating bugs in the history of SSL and TLS. The most widely used SSL/TLS implementation (OpenSSL) was found to have a buffer overread bug, allowing anyone to send a specially crafted message to any OpenSSL server and receive a dump of its memory, often revealing its long-term private keys.**

Yet, if a CA truly misbehaves, it can decide not to revoke malicious certificates, or not to report them. The problem is that we are blindly trusting a non-negligible number of actors (the CAs) to do the right thing. To solve this issue at scale, **Certificate Transparency** was proposed in 2012 by Google.

The idea behind certificate transparency is to force CAs to add each certificate they issue to a giant log of certificates for everyone to see. To do this, browsers like Chrome now reject certificates if they do not include proofs of inclusion in a public log. This transparency allows you to check if a certificate was wrongly issued for a domain you own (there should be no other certificates than the ones you requested).

Note that Certificate Transparency relies on people monitoring logs to catch bad actors **after the**

fact (and that is, if you're looking), as well as CAs to react and revoke mis-issued certificates once detected, or browsers to remove misbehaving CAs from their trust stores. Certificate Transparency is thus not as powerful as certificate/public key pinning which mitigate CA misbehaviors.

9.4 Other secure transport protocols

You've now learned about TLS, which is the most popular protocol to encrypt communications. You're not done yet though. TLS is not the only one in the class. Many other protocols exist, and you might most likely be using them already.

Yet, most of them are TLS-like protocols customized to support their specific use case. This is the case, for example, with:

- **Secure Shell (SSH)**, the most widely used protocol and application to securely connect to a remote terminal on a different machine.
- **Wi-Fi Protected Access (WPA)**, the most popular protocol to connect devices to private network access points or the internet.
- **IPSec**, one of the most popular virtual network protocols (VPNs) used to connect different private networks together. It is mostly used by companies to link different office networks. As its name indicates, it acts at the IP layer and is often found in routers, firewalls, and other network appliances. Another popular VPN is **OpenVPN** which makes direct use of TLS.

All of these protocols typically reimplement the handshake/post-handshake paradigm, and sparkle some of their own flavors to it. Re-inventing the wheel is not without issues, as for example, several of the Wi-Fi protocols have been broken.

To finish this chapter, I want to introduce you to a much more modern alternative to TLS: the **Noise Protocol Framework**.

9.5 The Noise protocol framework: a modern alternative to TLS

TLS is now quite mature, and considered a solid solutions in most cases due to the attention it gets, yet it adds quite a lot of overhead to applications that makes use of it due to historical reasons, backward compatibility constraints, and overall complexity.

Indeed, in many scenarios where you are in control of all endpoints, you might not need all of the features that TLS has to offer.

The next best solution is called the **Noise Protocol Framework**.

The Noise Protocol Framework removes the runtime complexity of TLS by avoiding all negotiation in the handshake. A client and a server running noise follow a linear protocol that does not branch. Contrast this to TLS which can take many different paths depending on

information contained in the different handshake messages. What Noise does is that it pushes all the complexity to the design phase. Developers who want to use the Noise protocol framework must decide what ad-hoc instantiation of the framework they want their application to use. (This is why it is called a framework and not a protocol.)

In such, they must first decide what cryptographic algorithms will be used, what side of the connection is authenticated, if any pre-shared key is used, and so on. After that, the protocol is implemented and turns into a rigid series of messages (which can be a problem if one needs to update the protocol later on, while maintaining backward compatibility with devices that cannot be updated).

9.5.1 The many handshakes of Noise

The Noise Protocol Framework offers different **handshake patterns** you can choose from. Handshake patterns typically come with a name that indicates what is going on.

For example, the **IK** handshake pattern indicates that the client's public key is sent as part of the handshake (the first I stands for "immediate"), and that the server's public key is known to the client in advance (the K stands for "known").

Once a handshake pattern has been chosen, applications making use of it will never attempt to perform any of the other possible handshake patterns. As opposed to TLS, this makes Noise a very simple and linear protocol in practice.

In the rest of this section, I will use a handshake pattern called *NN* to explain how Noise works, as it is quite simple (but insecure, since the two *N* indicate no authentication on both sides).

In Noise's lingo the pattern is written like this:

```
NN:
  -> e
  <- e, ee
```

Each line represents a message pattern, and the arrow indicates the direction of the message. Each message pattern is a succession of tokens (here there are only two: `e` and `ee`) that dictates what both sides of the connection have to do.

`e` means that the client must generate an ephemeral key pair and send the public key to the server. The server interprets this message differently: it must receive an ephemeral public key and store it.

`e, ee` means that the server must generate an ephemeral key pair and send the public key to the client, then it must do a Diffie-Hellman key exchange with the client's ephemeral (the first `e`) and its own ephemeral (the second `e`). On the other hand, the client must receive an ephemeral public key from the server, and use it to do a Diffie-Hellman key exchange as well.

> **NOTE** Noise uses a combination of defined tokens in order to specify different types of handshake. For example, the `s` token means a static key (another word for long-term key) as opposed to an ephemeral key, and the token `es` means that both participants must perform a Diffie-Hellman key exchange using the client's ephemeral key and the server's static key.

There's more to it: at the end of each message pattern (`e` and `e, ee`), the sender also gets to transmit a payload. If a Diffie-Hellman key exchange has happened previously (not the case in the first message pattern `e`), the payload is encrypted and authenticated.

At the end of the handshake both participants derive a set of symmetric keys and start encrypting communications, similarly to TLS.

9.5.2 A handshake with Noise

One particularity of Noise is that it continuously authenticates its handshake transcript. To achieve this, both sides maintain two variables: a hash `h` and a chaining key `ck`. Each handshake message sent or received is hashed with the previous `h` value. I illustrate this in figure 9.10.

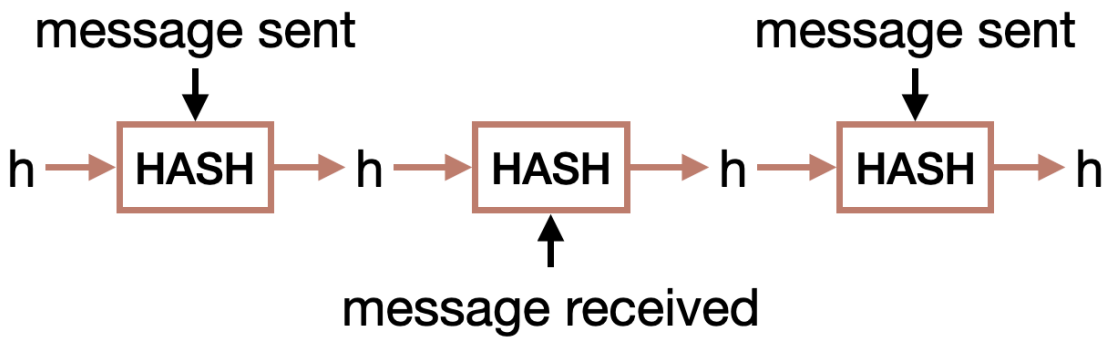


Figure 9.10 In the Noise Protocol Framework, each side of the connection keeps track of a digest `h` of all messages that have been sent and received during the handshake. Whenever a message is sent encrypted with an authenticated encryption with associated data algorithm, the current `h` value is used as associated data in order to authenticate the handshake up to this point.

At the end of each message pattern, if the payload is sent encrypted, it also authenticates the `h` value using the associated data field of the AEAD algorithm used (I covered this in chapter 4).

This allows Noise to continuously verify that both sides of the connection are seeing the exact same series of messages and in the same order.

In addition, every time a Diffie-Hellman key exchange happens (several can happen during a handshake), its output is fed along with the previous chaining key `ck` to HKDF in order to derive a new chaining key and a new set of symmetric keys to use for authenticating and encrypting

subsequent messages. I illustrate this in figure 9.11.

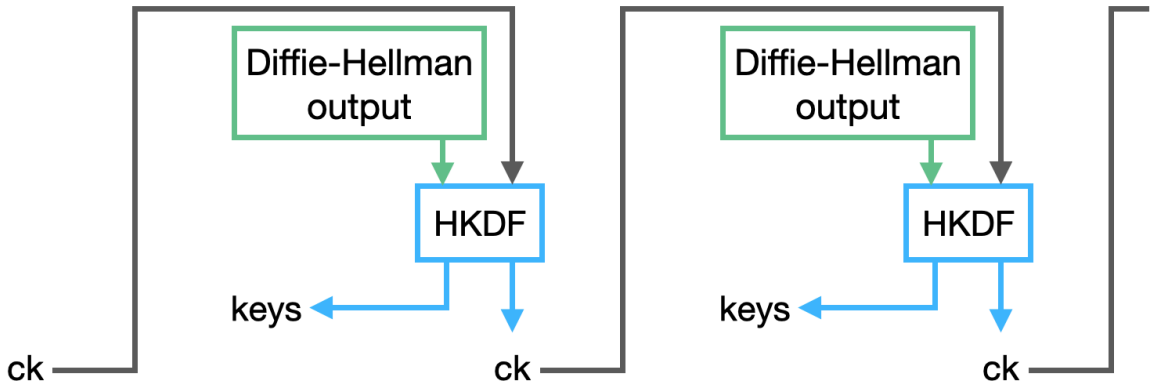


Figure 9.11 In the Noise Protocol Framework, each side of the connection keeps track of a chaining key ck. This value is used to derive a new chaining key and new encryption keys to be used in the protocol every time a Diffie-Hellman key exchange is performed.

This makes Noise a very simple protocol at run-time: there is no branching and both sides of the connection simply do what they need to do. Libraries implementing Noise are also extremely simple, and end up being a few hundred lines compared to hundreds of thousands of lines for TLS libraries.

While Noise is more complex to use, and will require developers who understand how Noise works to integrate it into an application, it is a very strong alternative to TLS.

9.6 Summary

- Transport Layer Security (TLS) is a secure transport protocol to encrypt communications between machines. It was previously called Secure Sockets Layer (SSL) and is sometimes still referred to as SSL.
- TLS works on top of TCP and is used every day to protect connections between browsers, web servers, mobile applications, and so on.
- To protect sessions on top of UDP, TLS has a variant called Datagram Transport Layer Security (DTLS) that works with UDP.
- TLS, and most other transport security protocols, have a handshake phase in which the secure negotiation is created, and a post-handshake phase in which communications are encrypted using keys derived from the first phase.
- To avoid delegating too much trust to the web public key infrastructure, applications making use of TLS can use certificate and public key pinning to only allow secure communications with specific certificates or public keys.
- As a defense-in-depth measure, systems can implement certificate revocation (to remove compromised certificates) and monitoring (to detect compromised certificates or CAs).
- In order to avoid TLS' complexity and size, and if you control both sides of the connection, you can use the Noise Protocol framework.
- To use the Noise protocol framework, one must decide what variant of handshake they want to use when designing the protocol. Due to this, it is much simpler and secure than TLS, but less flexible.

10 End-to-end encryption

This chapter covers

- The importance of end-to-end encryption for companies and users.
- The different attempts at solving email encryption.
- How end-to-end encryption is changing the landscape of messaging.

Chapter 9 explained transport security via protocols like TLS and Noise. At the same time, I spent quite some time explaining where trust was rooted on the web: hundreds of certificate authorities trusted by your browser and operating system. While not perfect, this system has worked so far for the web, which is a complex network of participants who know nothing of each other.

This problem of finding ways to trust others (and their public keys), and making it scale, is at the center of real-world cryptography. A famous cryptographer was once heard saying "symmetric crypto is solved" to describe a field of research that had overstayed its welcome. And for the most part the statement was true. We seldom have issues encrypting communications, and we have strong confidence in the current encryption algorithms we use. Most engineering challenges when it comes to encryption are not about the algorithms themselves anymore, but about who Alice and Bob are, and how to prove it.

Cryptography does not provide one solution to trust, but many different ones that are more or less practical depending on the context. In this chapter, I will survey some of the different techniques that people and applications have used to create trust between users.

10.1 Why end-to-end encryption?

This chapter starts with a "why" instead of a "what." This is because end-to-end encryption (often abbreviated as **e2e encryption**) is a concept more than a cryptographic protocol, a concept of securing communications between two (or more) participants across an adversarial path.

I started this book with a simple example: Alice wanted to send a message to Bob without anyone in the middle being able to see it. Nowadays, many applications like email and messaging exist to connect users, and most of them seldom encrypt messages from soup to nuts.

You might ask: isn't TLS enough? In theory it could be. You've learned in chapter 9 that TLS is used in many places to secure communications. But end-to-end encryption is a concept that involves **actual human beings**. In contrast, TLS is most often used by systems that are by design men-in-the-middle (see figure 10.1). In these, TLS is only used to protect the communications between a central server and its users, allowing the server to see everything. Effectively these man-in-the-middle servers that sit in-between users, and that that are necessary for the application to function, are **trusted third parties** of the protocol. That is to say, we have to trust these parts of the system in order for the protocol to be considered secure (spoiler alert: that's not a great protocol).

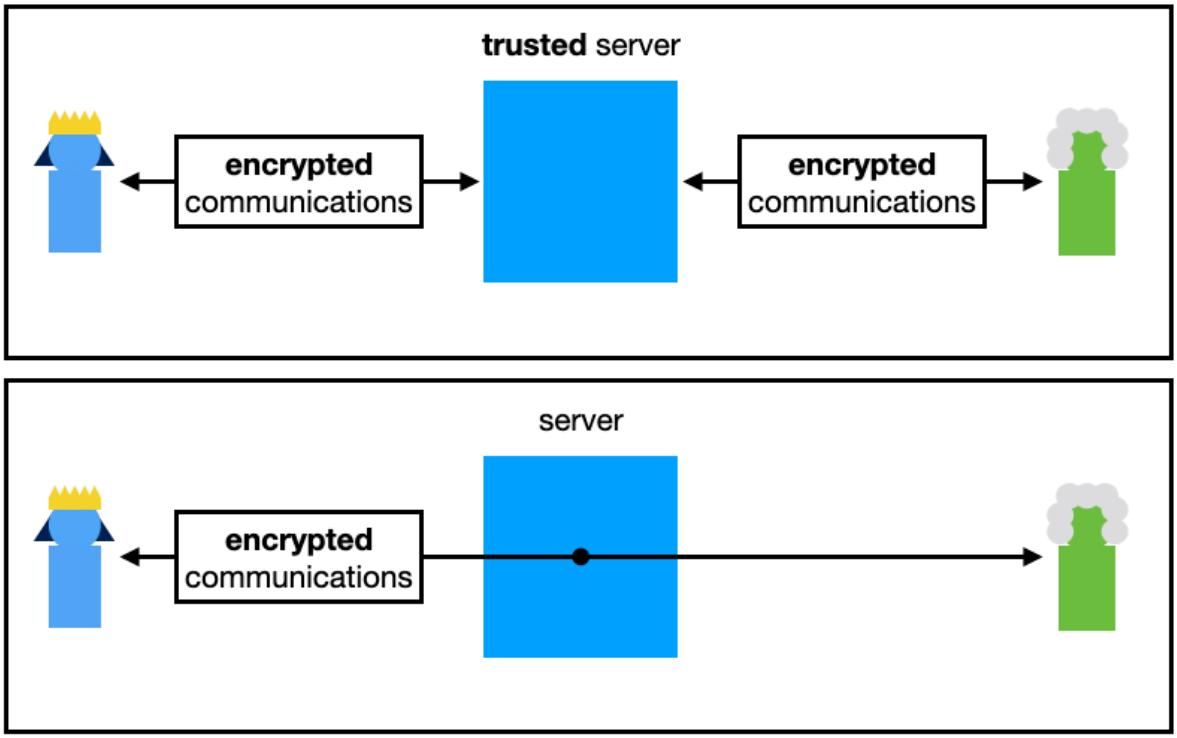


Figure 10.1 In most systems (top diagram), a central server is used to relay messages between users. A secure connection is usually established between a user and the central server (which can thus see all user messages). A protocol providing end-to-end encryption (bottom diagram) encrypts communications from one user up to its intended recipient, preventing any server in the middle from observing mssages in clear.

Sometimes worse topologies that increase the attack surface exist. Communications between a user and a server can go through several hoops, machines often referred to as middleboxes that end the TLS connection earlier (we say that they terminate TLS) and either forward the traffic in clear to, or start another TLS connection with, the next hoop. Perhaps this is done in order to better filter traffic, or balance connections geographically for optimization purposes, or to intercept, record, and spy on traffic.

Many cases exist where such interception (deliberate or not) happened, and of course many more that we don't know about. In 2015 Lenovo was caught selling laptops with pre-installed custom certificate authorities (covered in chapter 9) and software. The software was man-in-the-middle'ing HTTPS connections (using Lenovo's certificate authorities) and injecting ads into web pages. More seriously, large countries like China and Russia have been caught redirecting traffic on the internet (making it pass through their network) in order to intercept and observe connections.[8] In 2013, Edward Snowden leaked a massive number of documents from the NSA showing the abuses of many governments (not just the US) in spying on their people's communications by intercepting the internet cables that link the world together.[9]

Owning and seeing user data is also a liability for companies. As I've mentioned many times in this book, breaches and hacks happen way too often and can be devastating for the credibiility of a company. From a legal standpoint, laws like the General Data Protection Regulation (GDPR) can end up costing organizations a lot of money. British Airways was fine £183.39 million after a data breach that happened in 2019. Governement requests like the infamous National Security Letters (NSLs) that sometimes prevent companies and people involved from sharing them (so-called gag orders) can be seen as additional cost and stress to an organization too, that is unless you have nothing much to share.

Bottom line, if you're using a popular online application, chances are that one or more governments already have access to everything you wrote or uploaded there. Depending on an application's **threat model** (what the application wants to protect against), or the threat model of an application's most vulnerable users, end-to-end encryption plays a major role in ensuring confidentiality and privacy of end-users.

This chapter will go over different techniques and protocols that have been created in order to create trust between people. In particular, you will learn about how email encryption works today, and how secure messaging is changing the landscape of end-to-end encrypted communications.

10.2 A root of trust nowhere to be found

The simplest scenario for end-to-end encryption is the following:

Alice wants to send an encrypted file to Bob over the internet.

With all the cryptographic algorithms you've learned about in the first chapter of this book, you can probably think of a way to do this if you've read chapter 6. For example:

1. Bob sends his public key to Alice.
2. Alice encrypts the file with Bob's public key, and sends it to Bob.

Perhaps Alice and Bob can meet in real life, or use another secure channel they already share, to exchange the public key in the first message. If this is possible, we say that they have an **out-of-band** way of creating trust.

This is not always the case though. You can imagine me including my own public key in this book, and asking you to use it to send me an encrypted message at some email address. Who says my editor did not replace the public key with theirs?

Same for Alice, how does she figure out if the public key she received truly is Bob's public key? It's possible that someone in the middle could have tampered with the first message as illustrated in figure 10.2.

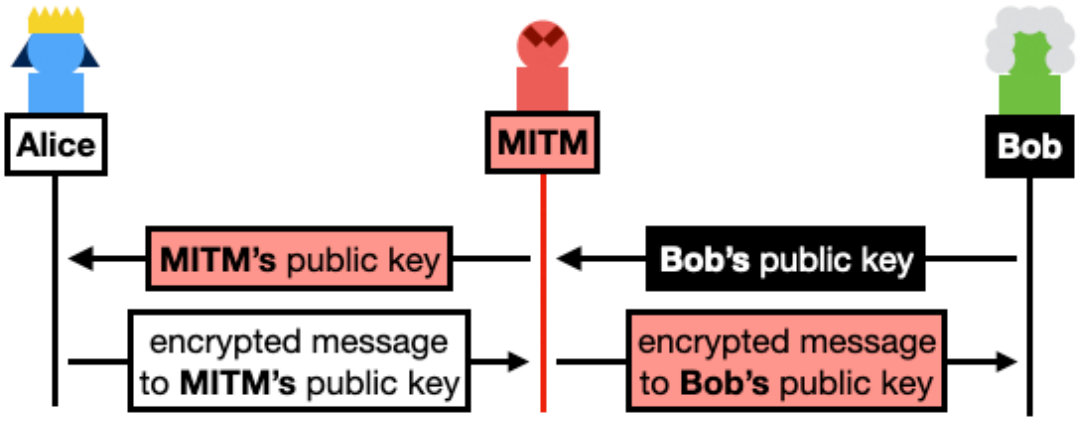


Figure 10.2 Bob sends his public key to Alice so that she can use it to encrypt her messages to him. As nobody is authenticated, a man-in-the-middle attacker can simply replace Bob's public key with their own.

A painful realization is that **there is no real solution to this trust issue**. As you will see in this chapter cryptography has no real answer to the trust issue, instead it provides different solutions to help in different scenarios.

> *This whole business of protecting public keys from tampering is the single most difficult problem in practical public key applications. It is the 'Achilles heel' of public key cryptography, and a lot of software complexity is tied up in solving this one problem.*
>
> *– PGP User's Guide Volume I: Essential Topics*

The reason why there is no true solution is that we are trying to bridge reality (real human

beings) to a theoretical cryptographic protocol.

Going back to our simple setup where Alice wants to send a file to Bob, and assuming that their untrusted connection is all they have, they have somewhat of an impossible trust issue at hand. Alice has no good way of knowing for sure what truly is Bob's public key. It's a chicken and egg type of problem.

Yet, let me point out that if no malicious **active** man-in-the-middle attacker is here to replace Bob's public key in the first message, then the protocol is safe. Even if the messages are being **passively** recorded, it is too late for an attacker to come after the fact to decrypt the second message.

Of course, relying on the fact that your chances of being actively man-in-the-middled are "not too high" is not the best way to do cryptography, but we unfortunately often do not have a way to avoid this. For example, Google Chrome ships with a set of certificate authorities that you choose to trust, but how do you obtain Chrome in the first place? Perhaps you used the default browser of your operating system, which relies on its own set of certificate authorities. But where did that come from? From the laptop you bought. But where did this laptop come from?

As you can quickly see, it's **turtles all the way down**. At some point, you will have to trust that something was done right. A threat model typically chooses to stop addressing issues after a specific turtle, and considers that any turtle further down is out-of-scope.

This is why the rest of the chapter will assume that you have a secure way to obtain some **root of trust**. All systems based on cryptography work by relying on a root of trust, something that a protocol can build security on top of. A root of trust can be a secret or a public value that we start the protocol with, or an out-of-band channel that we can use to obtain these.

10.3 The failure of encrypted email

Email was created as, and is still today, an unencrypted protocol. We can only blame a time where security was second thought. Email encryption started to become more than just an idea after the release of a tool called **Pretty Good Privacy (PGP)** in 1991. At the time, the creator of PGP Phil Zimmermann decided to release PGP in reaction to a bill that almost came to law earlier in the same year. The bill would have allowed the US government to obtain all voice and text communications from any electronic communication company and manufacturer. The protocol was finally standardized in RFC 2440 as **OpenPGP** in 1998, and caught traction with the release of the open source implementation GNU Privacy Guard (GPG) around the same time. [10] Today, GPG is still the main implementation and people interchangeably use the terms GPG and PGP to pretty much mean the same thing.

10.3.1 PGP or GPG? And how does it work?

PGP, or OpenPGP, works by simply making use of hybrid encryption (covered in chapter 6). The details are in RFC 4880, the last version of OpenPGP, and can be simplified to these steps:

1. The sender creates an email. At this point the email's content is usually compressed before it is encrypted (do you know why it is not compressed after encryption?)
2. The OpenPGP implementation generates a random symmetric key and symmetrically encrypts the email using the symmetric key.
3. The symmetric key is asymmetrically encrypted to all the recipients' public keys (using techniques you've learned in chapter 6).
4. The email body is replaced with the encrypted version of the message and sent to all recipients along with their respective encrypted symmetric key.
5. To decrypt an email, a recipient uses their private key to decrypt the symmetric key, then decrypts the content of the email using the decrypted symmetric key.

Note that OpenPGP also defines how an email can be signed (in order to authenticate the sender). To do this, the plaintext email's body is hashed and then signed using the sender's private key. The signature is then added to the message before being encrypted in step 2. Finally, so that the recipient can figure out what public key to use to verify the signature, the sender's public key is sent along the encrypted email in step 4.

I illustrate the PGP flow in figure 10.3.

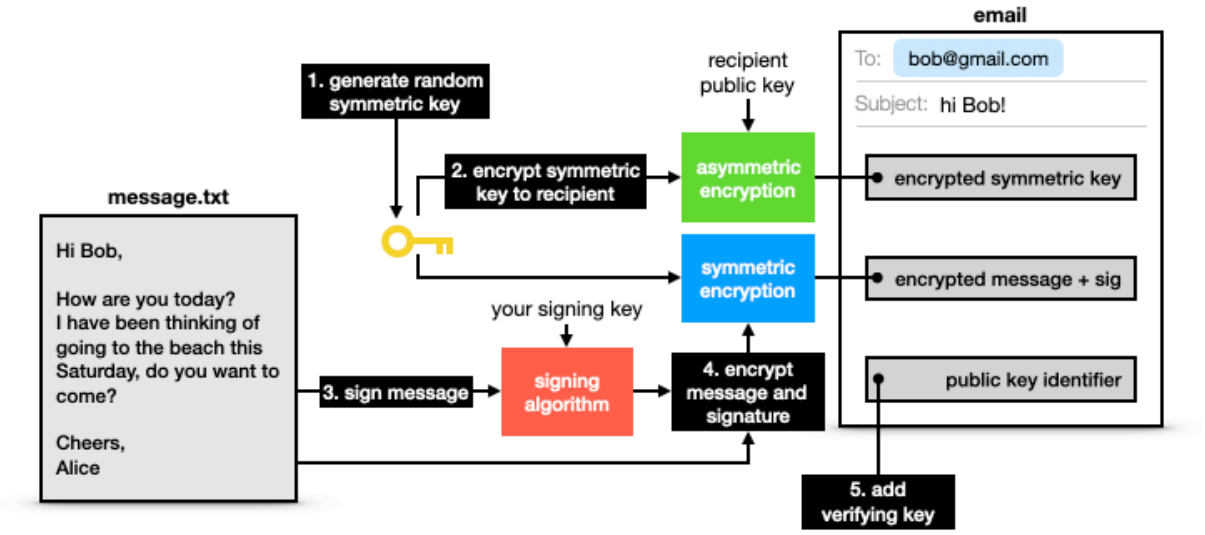


Figure 10.3 PGP's goal is to encrypt and sign messages. When integrated to email clients it does not care about hiding the subject or other metadata.

There's nothing inherently wrong with this design at first sight. It seems to prevent man-in-the-middle attackers from seeing your emails's content (although the subject and other email headers are not encrypted).

> **NOTE** It is important to note that cryptography can only go so far to hide metadata. In privacy-conscious applications, metadata is a big problem and can in the worst cases de-anonymize you! For example, in end-to-end encrypted protocols you might not be able to decrypt messages between users, but you can probably tell what their IP addresses are, and what is the length of the messages they send and receive, and who they commonly talk to (their social graphs), and so on. A lot of engineering is put into hiding this type of metadata.

Yet, in the details PGP is actually quite bad.

The OpenPGP standard and its main implementation GPG make use of old algorithms and ways. The most critical issue is that encryption is not authenticated, which means that anyone intercepting an email that hasn't been signed might be able to tamper with the encrypted content to some degree depending on the exact encryption algorithm used. For this reason alone I would not recommend anyone to use PGP today.

A surprising flaw of PGP comes from the fact that the signing and encryption operations are composed without care. In 2001 Don Davis pointed out that because of this naive composition of cryptographic algorithms, one can reencrypt a signed email they received to another recipient. Effectively allowing Bob to send you the email Alice sent him, like you were the intended recipient.

If you're wondering, signing the ciphertext instead of the plaintext is still flawed, as one could then simply remove the signature that comes with the ciphertext and add their own signature instead. In effect, Bob could pretend that they sent you an email that was actually coming from Alice. I recapitulate these two signing issues in figure 10.4. By the way, can you think of a unambiguous way of signing a message?

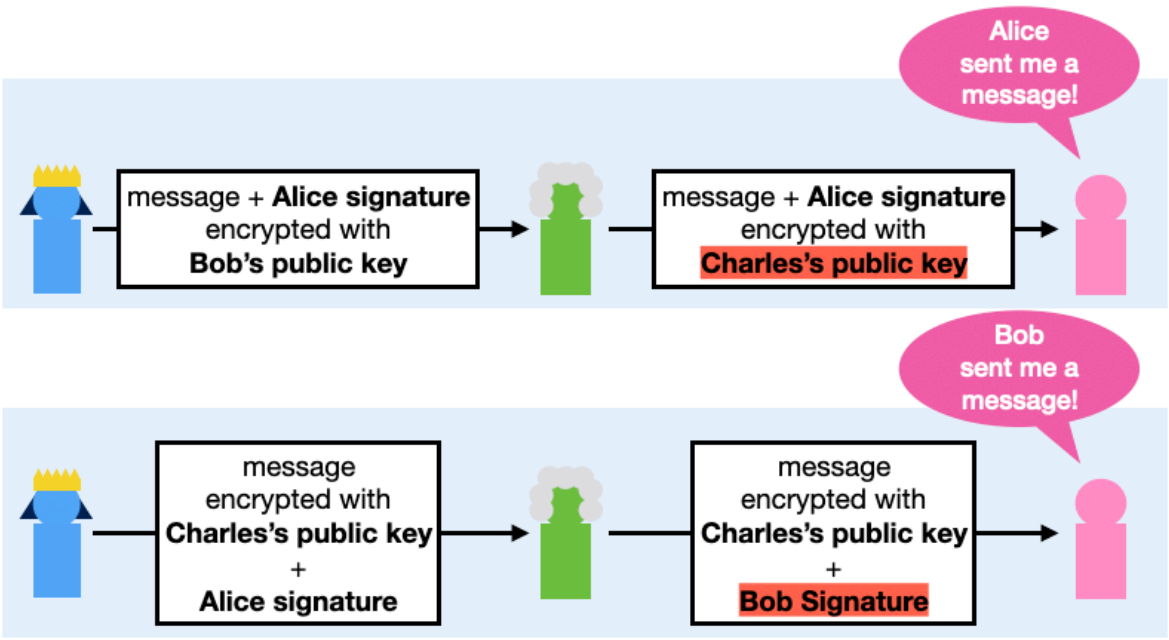


Figure 10.4 In the top diagram, Alice encrypts a message and signature (over the message) with Bob's public key. Bob can reencrypt this message to Charle's who might believe that it was intended for him. This is the PGP flow. In the bottom diagram, this time Alice encrypts a message to Charles. She also signs the encrypted message instead of the plaintext content. Bob who intercepts the encrypted message can replace the signature with its own, fooling Charles into thinking that he wrote the content of the message.

The cherry on the cake is that the algorithm does not provide forward secrecy by default. As a reminder: without forward secrecy, a compromise of your private key implies that all previous emails sent to you encrypted under that key can now be decrypted. You can still force forward secrecy by changing your PGP key, but this process is not straightforward (you can, for example, sign your new key with your older key) and most users just don't bother.

So to recap:

- PGP uses old cryptographic algorithms.
- PGP does not have authenticated encryption, and is thus not secure if used without signatures.
- Due to bad design, receiving a signed message doesn't necessarily mean we were the intended recipient.
- There is no forward secrecy by default.

10.3.2 Scaling trust between users with the web of trust

So why am I really talking about PGP here? Well, there is something interesting about PGP that I haven't talked about yet: how do you obtain other people's public keys?

The answer is that in PGP, **you build trust yourself**.

OK what does this mean? Imagine that you install GPG today and decide that you want to

encrypt some messages to your friends. To start, you must first find a secure way to obtain your friends' PGP public keys. Meeting them in real life is one sure way to do it. So you meet, you copy their public keys on a piece of paper, and then you type those keys back into your laptop at home. Now you can send your friends signed and encrypted messages with OpenPGP.

But this is tedious. Do you have to do this for every person you want to email? Of course not.

Let's take the following scenario:

- You have obtained Bob's public key in real life and thus you trust it.
- You do not have Mark's public key, but Bob does and trusts it.

Take a moment here to think about what you could be doing to trust Mark's public key.

Bob can simply sign Mark's key, showing you that he trusts the association between the public key and Mark's email. If you trust Bob, you can now trust Mark's public key and add it to your repertoire.

And this is all there is to the concept of decentralized trust behind PGP. It is called **the web of trust (WoT)**.

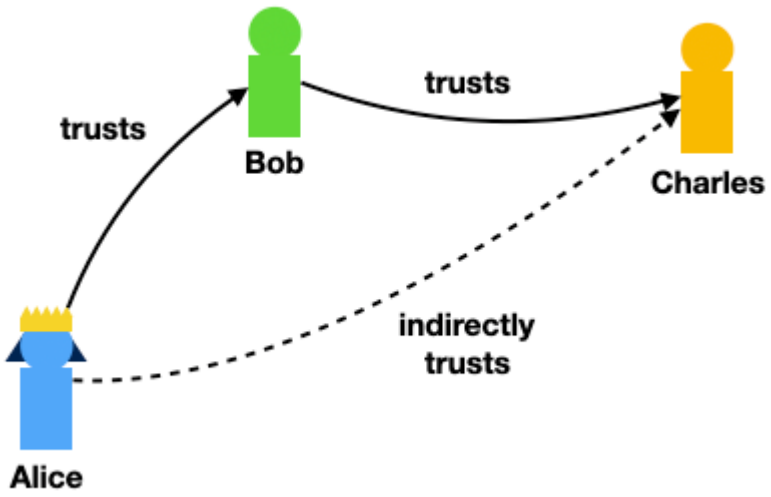


Figure 10.5 The web of trust is the concept that users can transitively trust other users by relying on signatures. In this diagram we can see that Alice trusts Bob who trusts Charles. Alice can use Bob's signature over Charles identity and public key to trust Charles as well.

You will sometimes see "key parties" at conferences, where people meet in real life and sign their respective public keys. But most of that is role-playing, and in practice very few people rely on the web of trust to enlarge their PGP circle.

10.3.3 Key discovery is a real issue

PGP did try other ways to solve the issue of discovering public keys: key registries. The concept is pretty simple, publish your PGP public key (and associated signatures from others that attest your identity) on some public list so that people can find it. In practice this doesn't work as anyone can publish a signature under your email.

In some settings, we can relax our threat model and allow for a trusted authority to attest identites and public keys. Think of a company managing their employees' emails, for example.

In 1995, the RSA company proposed **Secure/Multipurpose Internet Mail Extensions (S/MIME)** as an extension to the MIME format (which itself is an extension to the email standard), and as an alternative to PGP. S/MIME, standardized in RFC 5751, took an interesting departure from the web of trust by using a public key infrastructure to build trust. That is pretty much the only conceptual difference that S/MIME has with PGP.

As companies have processes in place to onboard and offboard employees, it makes sense for them to start using protocols like S/MIME in order to bootstrap trust in their internal email ecosystem.

It is important to note that both PGP and S/MIME are usually used over the **Simple Mail Transfer Protocol (SMTP)** which is the protocol used today for emails. These protocols were also invented later, and for this reason their integration with SMTP and email clients is far from perfect. For example, only the body of an email is encrypted, not the subject or any of the other email headers.

S/MIME, like PGP, is also quite an old protocol that uses outdated cryptography and practices. Like PGP, it does not offer authenticated encryption.

Recent research on integration of both protocols in email clients showed that most of them were vulnerable to exfiltration attacks where an attacker who can observe encrypted emails can retrieve their content by sending tampered versions to the recipients.[11]

Furthermore, this does not even matter as today most email is still sent unencrypted. PGP has proven to be quite hard to use for normal as well as advanced users who need to understand the many subtleties and processes of PGP's key management. And I'm not even talking about poor integration with email clients, where users can still respond to an encrypted email without encryption, potentially quoting the whole thread in cleartext.

> *In the 1990s, I was excited about the future, and I dreamed of a world where everyone would install GPG. Now I'm still excited about the future, but I dream of a world where I can uninstall it.*
>
> *– Moxie Marlinspike GPG And Me (2015)*

For these reasons, PGP has slowly been losing support (for example Golang removed support for PGP from its standard library in 2019), while more and more real-world cryptography applications are aiming at replacing PGP and solving its usability problems.

Today, it is hard to argue that email encryption will ever be a thing.

> *If messages can be sent in plaintext, they will be sent in plaintext.*
>
> *Email is end-to-end unencrypted by default. The foundations of electronic mail are plaintext. All mainstream email software expects plaintext. In meaningful ways, the Internet email system is simply designed not to be encrypted.*
>
> *– Thomas Ptatcek - Stop Using Encrypted Email (2020)*

10.3.4 If not PGP, then what?

I spent some pages to talk about how a simple design like PGP can fail in a lot of different and surprising ways in practice.

I would recommend against using PGP.

While email encryption is still an unsolved problem, alternatives are being developed to replace different use cases of PGP.

Saltpack is a similar protocol and message format to PGP. It attempts to fix some of the PGP flaws I've talked about. In 20201, Saltpack's main implementations are keybase (https://keybase.io) and keys.pub (https://keys.pub). (See figure 10.6 for an illustration of the keys.pub tool.)

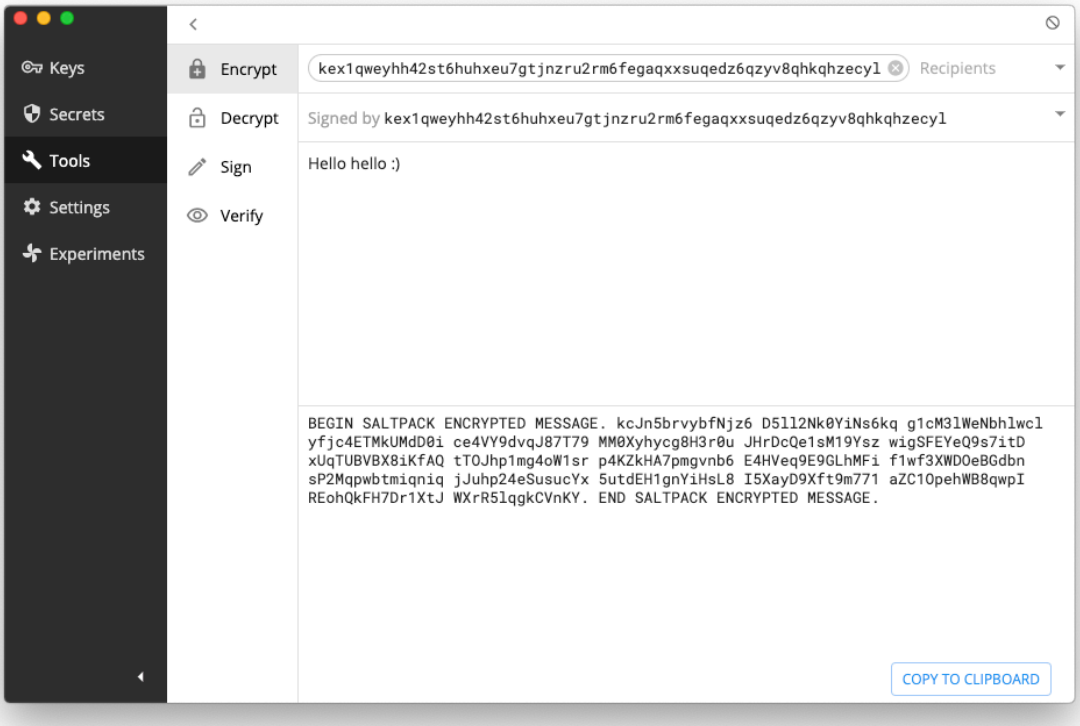


Figure 10.6 keys.pub is a native desktop application implementing the saltpack protocol. You can use it to import other people's public keys, and encrypt and sign messages to them.

These implementations have all moved away from the web of trust, and allow users to broadcast their public keys on different social networks in order to instill their identity into their public keys (as illustrated in figure 10.7). PGP could obviously not have thought about this key discovery mechanism, as it predates the boom of social networks.

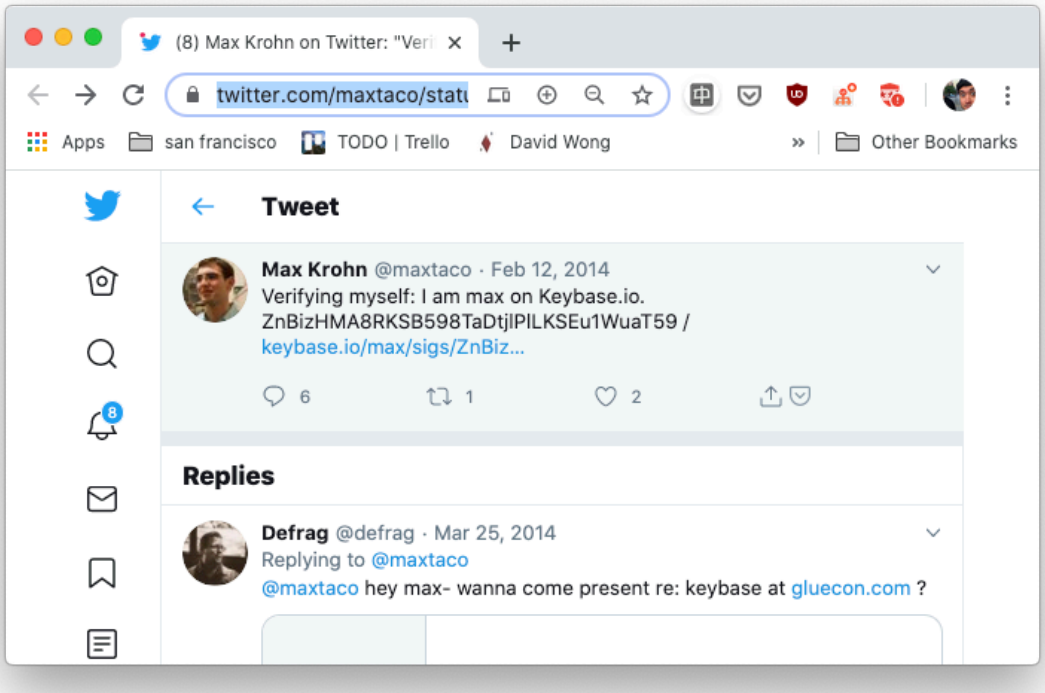


Figure 10.7 A keybase user broadcasting their public key on the Twitter social network. Allowing other users to obtain additional proof that his identity is linked to a specific public key.

On the other hand, most secure communication nowadays is far from a one-time message, and the use of these tools is less and less relevant. In the next section I talk about **secure messaging**, one of the fields that aim at replacing the communication aspect of PGP.

10.4 Secure messaging, a modern look at end-to-end encryption with Signal

In 2004, **Off-The-Record (OTR)** was introduced in white paper entitled "Off-the-Record Communication, or, Why Not To Use PGP." Unlike PGP or S/MIME, OTR is not used to encrypt emails but instead chat messages, specifically it extends a chat protocol called the Extensible Messaging and Presence Protocol (XMPP).

One of the particularity of OTR was **deniability**. A claim that recipients of your messages (and passive observers) cannot use messages you sent them in a court of justice. Since messages you send are authenticated and encrypted symmetrically with a key your recipient shares with you, they could have easily forged these messages themselves. By contrast, in PGP messages are signed and are thus the inverse of deniable: they are **non-repudiable**. To my knowledge, none of these properties have actually been tested in court.

In 2010 the Signal mobile phone application (then called TextSecure) was released, making use

of a newly created protocol called the **Signal protocol**. The Signal mobile application was a large departure from PGP, S/MIME, OTR and other protocols based on **federation** (no central entity is required for the network to run) by taking a centralized approach like most commercial products do. Yet Signal is a nonprofit organization that runs solely on grants and donations.

While Signal prevents interoperability with other servers, the Signal protocol is an open standard and has been adopted by many other messaging applications, including Google Allo (now defunct), WhatsApp, Facebook Messenger, Skype, and many others. The Signal protocol is truly a success story, transparently being used by billions of people, including journalists, targets of government surveillance, and your average joe.

It is interesting to look at how Signal works, because it attempts to fix many of the flaws that I've previously mentioned with PGP. In this section, I will go over each one of these interesting features of Signal:

1. How can we do better than the web of trust? Is there a way to upgrade the existing social graphs with end-to-end encryption? The answer of Signal is to use a **trust on first use (TOFU)** approach. TOFU allows users to blindly trust other users the first time they communicate, relying on this first insecure exchange to establish a long-lasting secure communication channel. Users are then free to check if the first exchange was man-in-the-middled by matching their session secret out-of-band.
2. How can we upgrade PGP to obtain forward secrecy every time we start a conversation with someone? The first part of the Signal protocol is like most secure transport protocols: it's a key exchange, but a particular one called **Extended Triple Diffie-Hellman (X3DH)**. More on that later.
3. How can we upgrade PGP to obtain forward secrecy for every single message. This is important as conversations between users can span years and a compromise at some point in time should not reveal years of communication. Signal addresses this with something called a **symmetric ratchet**.
4. What if two users' session secrets are compromised at some point in time? Is that game over? Can we also recover from that? Signal introduces a new security property called **post-compromise security** and addresses it with what they call a **Diffie-Hellman ratchet**.

Let's get started!

10.4.1 Trust but verify

One of email encryption's biggest failure was its reliance on PGP and the web of trust model to transform social graphs into secure social graphs. It is really rare today to see people signing each other PGP keys.

The way most people use applications like PGP, OTR, Signal, and so on. is to blindly trust a key the first time they see it, and to reject any future changes (as illustrated in figure 10.8). This way only the very first connection can be attacked, and this only by an active man-in-the-middle attacker.

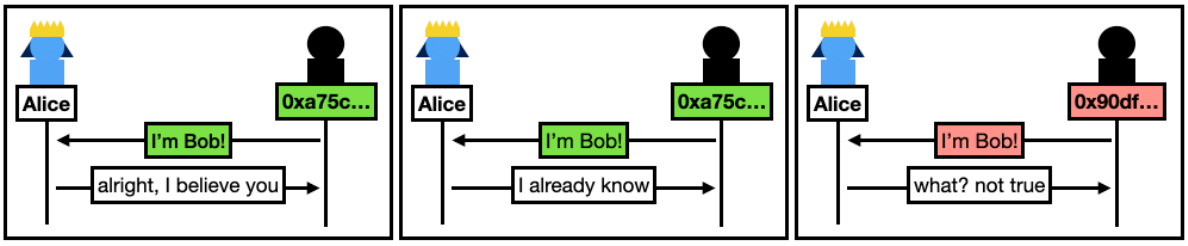


Figure 10.8 Trust On First Use (TOFU) allows Alice to trust her very first connection, but not subsequent connections if they don't exhibit the same public key. TOFU is an easy mechanism to build trust when the chances that the very first connection is actively man-in-the-middled are low. The association between a public key and the identity (here Bob) can also be verified after the fact out-of-band.

While TOFU is not the best security model, it is often the best we have, and has proven extremely useful. The Secure Shell (SSH) protocol, for example, is often used by trusting the server's public key during the first connection (see figure 10.9) and by rejecting any future change.

Figure 10.9 SSH clients use trust on first use. the first time you connect to an SSH server (see left picture) you have the option to blindly trust the association between the SSH server and the public key displayed. If the public key of the SSH server later changes (see right picture) your SSH client will prevent you from connecting to it.

While TOFU systems trust the first key they see, they still allow the user to later verify that the key is indeed the right one and catch any impersonation attempts. In real-world applications users typically compare **fingerprints**, which are most often hexadecimal representations of public keys. Sometimes hashes of the public keys are used to prevent disclosing what the public key is. This verification is of course done out-of-band. (If the SSH connection was compromised, then the verification would be compromised as well.)

> **NOTE** Of course, if users do not verify fingerprints then they can be man-in-the-middled without knowing it. But that is the kind of trade-off that real-world applications have to deal with if they want to bring end-to-end encryption at scale. Indeed, the failure of the web of trust shows that security-focused applications must keep usability in mind to get widely adopted.

In the Signal mobile application, a fingerprint between Alice and Bob is computed by:

- hashing Alice's identity key prepended to her username (a phone number in Signal)
- hashing Bob's identity key prepended to his username
- truncating both results and interpret them as a series of 6 numbers of 5 digits
- displaying the concatenation of the two series of 6 numbers to the user

I recapitulate this flow in figure 10.10.

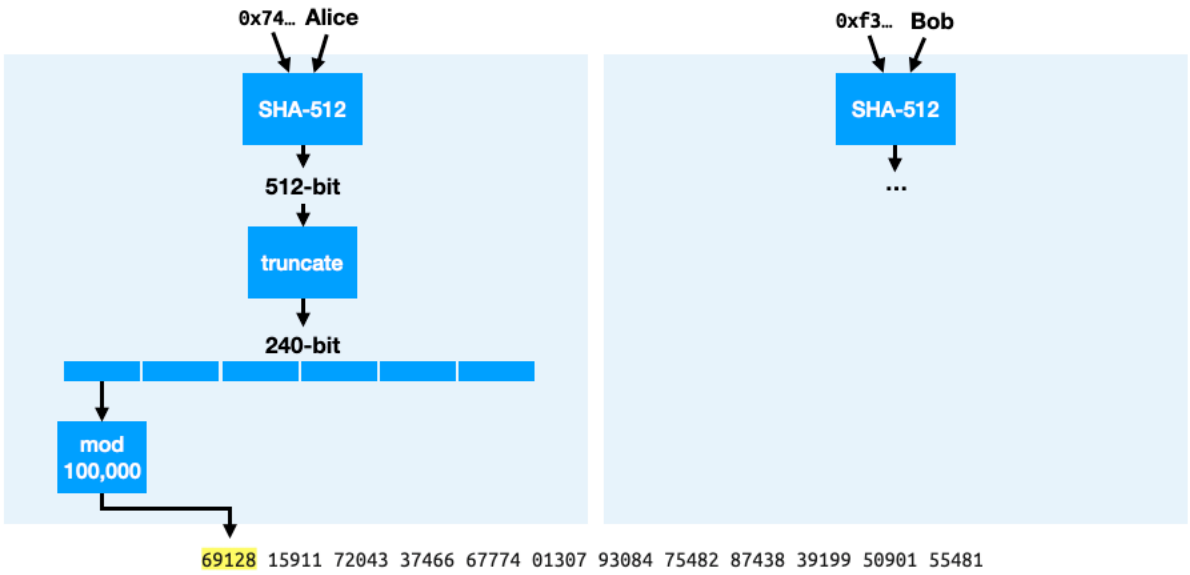


Figure 10.10 To compute a relationship's fingerprint, Signal uses SHA-512 to hash the identity key followed by the username of a peer, then truncate the result and interpret it as six 5-byte numbers taken modulo 100,000.

Applications like Signal make use of QR codes (see figure 10.11) to let users easily verify fingerprints more easily (as they can be pretty long).

Figure 10.11 In Signal, you can verify the authenticity and confidentiality of the connection you have with a friend by using a different channel (like the real life) to make sure the two fingerprints (called "safety numbers" in Signal) you and your friend have match. This can be done more easily via the use of QR code that encodes this information in a scannable format. Signal also hashes the session secret instead of the two users' public keys, allowing them to verify one large string instead of two.

Next, let's see how the Signal protocol works under the hood.

10.4.2 X3DH, the Signal protocol's handshake

Most secure messaging apps before Signal were **synchronous**. This meant that, for example, Alice wasn't able to start (or continue) an end-to-end encrypted conversation with Bob if Bob was not online. The Signal protocol on the other hand is **asynchronous**, like email, meaning that Alice can start (and continue) a conversation with people that are offline. Remember that **forward secrecy** (covered in chapter 9) means that a compromise of keys does not compromise previous sessions, and that forward secrecy usually means that the key exchanges are interactive (as both sides have to generate ephemeral Diffie-Hellman keypairs). In this section you will see how Signal uses non-interactive key exchanges (key exchanges where one side is potentially offline) that are still forward secure.

OK let's get going.

To start a conversation with Bob, Alice initiates a key exchange with him. Signal's key exchange is named **Extended Triple Diffie-Hellman (X3DH)** as it combines three (or more) Diffie-Hellman key exchanges into one. But before you learn how it works, you need to understand the three different types of Diffie-Hellman keys Signal uses:

- **Identity keys**. These are the long-term keys that represent the users. You can imagine that if Signal only used identity keys, then the scheme would be pretty similar to PGP and there would be no forward secrecy.
- **One-time prekeys**. In order to add forward secrecy to the key exchange, even when the recipient of a new conversation is not online, Signal has users upload multiple single-use public keys. These are simply ephemeral keys that are uploaded in advance, and are deleted after being used.
- **Signed prekeys**. We could stop here, but there's one edge case missing. Since the one-time prekeys that a user has uploaded can at some point be depleted, users also have to upload a medium-term public key that they sign (hence the name) and that they rotate periodically (for example, every week). This way, if no more ephemeral public keys (one-time prekeys) are available on the server under your username, one can still use your medium-term public key (signed prekey) to add forward secrecy up to the last time you changed your signed prekey.

This is enough to preview what the flow of a conversation creation in Signal looks like in figure 10.12.

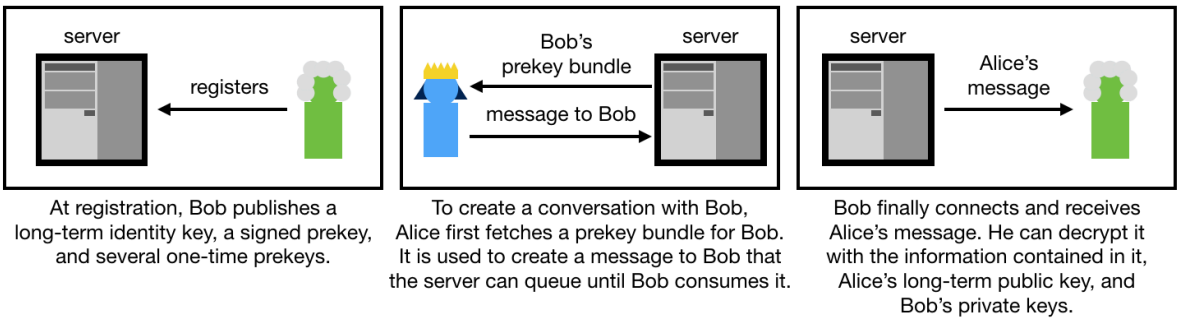


Figure 10.12 Signal's flow starts with a user registering with a number of public keys. If Alice wants to talk to user Bob, she first retrieves some public key information about the user (called a bundle of keys in the diagram), then performs an X3DH key exchange and creates an initial message using the output of the key exchange. Bob can perform the same on his side, after receipt of the message, to initialize the conversation.

Let's go over each of these steps in more depth.

First, a user registers by sending:

- one identity key
- one signed prekey and its signature
- a defined number of one-time prekeys

At this point, it is the responsibility of the user to periodically rotate the signed prekey and upload new one-time prekeys.

> **NOTE** **Signal makes use of the identity key to perform signatures (over signed prekeys) and key exchanges (during the X3DH key exchange). While I've warned against using the same key for different purposes, Signal has deliberately analyzed that in their case there should be no issue. This does not mean that this would work in your case, and with your key exchange algorithm, and so I would still advise against doing this.**

I recapitulate this flow in figure 10.13.

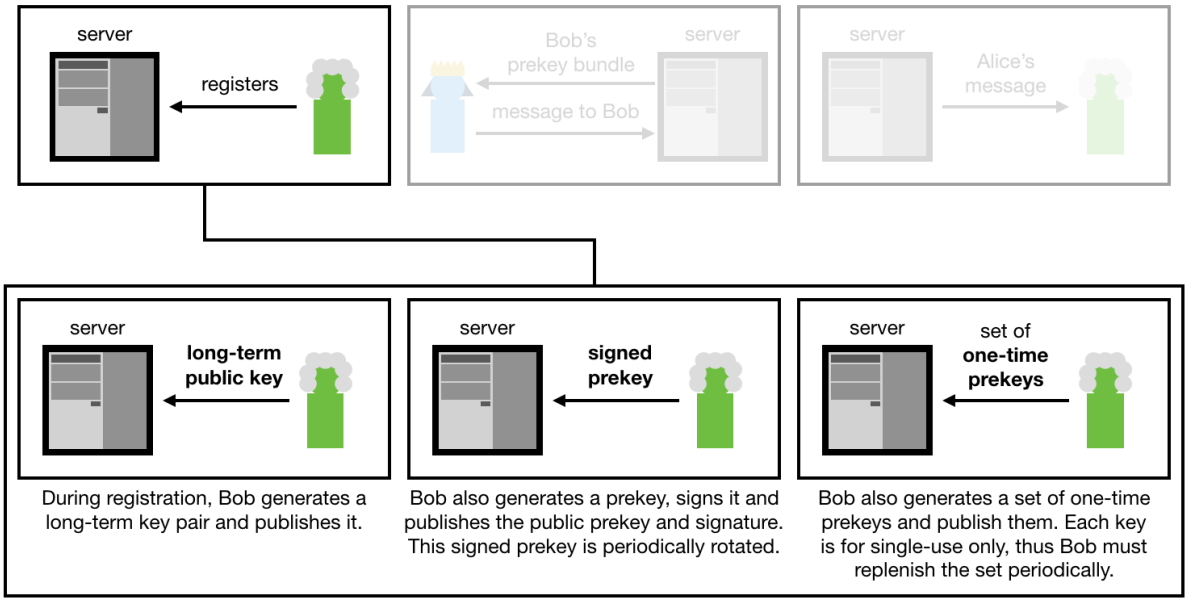


Figure 10.13 Building on figure 10.12. The first step is for a user to register by generating a number of Diffie-Hellman key pairs and sending the public parts to a central server.

Second, Alice starts a conversation with Bob by retrieving:

- Bob's identity key
- Bob's current signed prekey and its associated signature
- if there are still some: one of Bob's one-time prekeys (the server then deletes the one-time prekey sent to Alice)

Alice can then verify that the signature over the signed prekey is correct, and perform the X3DH handshake with:

- all of these public keys from Bob
- an ephemeral keypair that she generates for the occasion, in order to add forward secrecy
- her own identity key

The output of X3DH is then used in a post-X3DH protocol to encrypt her messages to Bob (more on that in the next section).

X3DH is composed of three (optionally four) Diffie-Hellman key exchanges grouped in one. They are Diffie-Hellman key exchanges between:

1. the identity key of Alice and the signed prekey of Bob
2. the ephemeral key of Alice and the identity key of Bob
3. the ephemeral key of Alice and the signed prekey of Bob
4. if Bob still has a one-time prekey available, his one-time prekey and the ephemeral key of Alice

The output of X3DH is the concatenation of all of these Diffie-Hellman key exchanges, passed to

a key derivation function (covered in chapter 8).

Different key exchanges provide different properties. 1 and 2 are here for mutual authentication, while 3 and 4 are here for forward secrecy. All of this is analyzed in more depth in the X3DH specification (https://signal.org/docs/specifications/x3dh/), which I encourage you to read if you want to know more (as it is very well written).

I recapitulate this flow in figure 10.14.

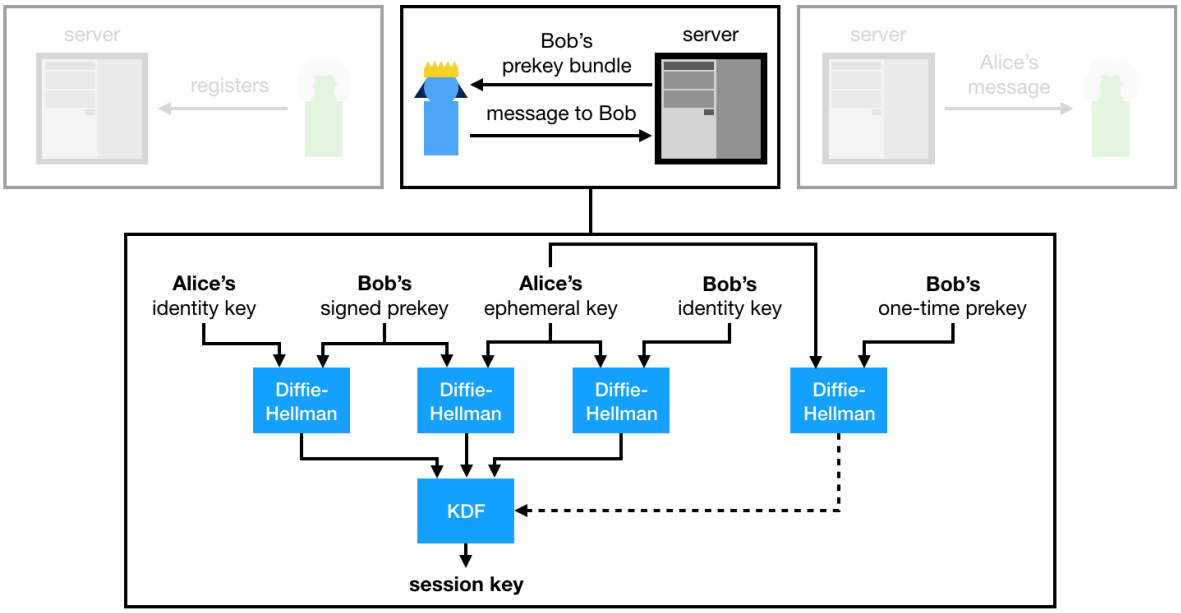


Figure 10.14 Building on figure 10.12. To send a message to Bob, Alice fetches a prekey bundle containing Bob's long-term key, Bob's signed prekey and optionally one of Bob's one-time prekey. After performing different key exchanges with the different keys, all outputs are concatenated and passed into a KDF to produce an output used in a subsequent post-X3DH protocol to encrypt messages to Bob.

Alice can then send to Bob her identity public key, the ephemeral public key she generated to start the conversation, and other relevant information (like which one-time prekey from Bob she used).

Bob receives the message and can perform the same X3DH key exchange with the public keys contained in it. Bob also gets rid of the one-time prekey of his that Alice decided to use in X3DH (if there was one).

I recapitulate this flow in figure 10.15.

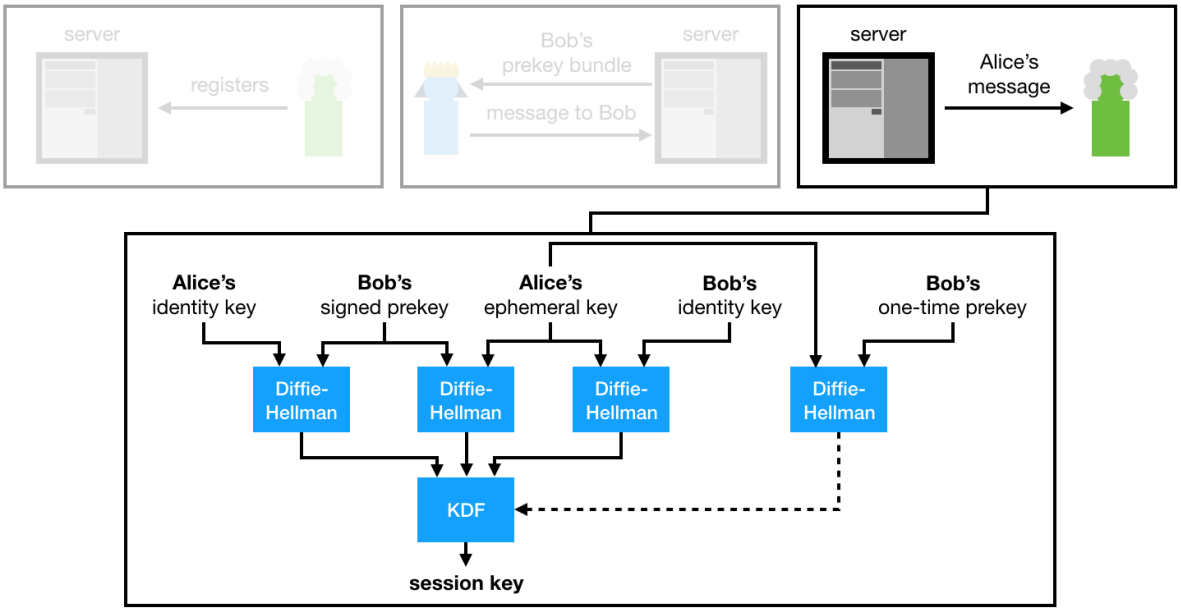


Figure 10.15 Building on figure 10.12. After receiving the initial message, Bob has all he needs to perform the same X3DH key exchange Alice had performed, deriving the same output used in the post-X3DH protocol.

What happens after X3DH is done? Let's see that next.

10.4.3 Double ratchet: Signal's post-handshake protocol

The post-X3DH phase lives as long as the two users do not delete their conversations or lose any of their keys. For this reason, and because Signal was designed with SMS conversations in mind (where the time between two messages might be counted in months), Signal introduces forward secrecy at the message level. In this section you will learn how this post-handshake protocol called the **Double Ratchet** works.

But imagine a simple post-X3DH protocol first. Alice and Bob could have taken the output of X3DH as a session key and use it to encrypt messages between them (as illustrated in figure 10.16).

Figure 10.16 Naively, a post-X3DH protocol could simply use the output of X3DH as a session key to encrypt messages between Alice and Bob.

We usually want to separate the keys used for different purposes though. So what we can do, is to use the output of X3DH as a seed (or root key according to the Double Ratchet specification)

to a KDF in order to derive two other keys. One key can be used for Alice to encrypt messages to Bob, and the other for Bob to encrypt messages to Alice. I illustrate this in figure 10.17.

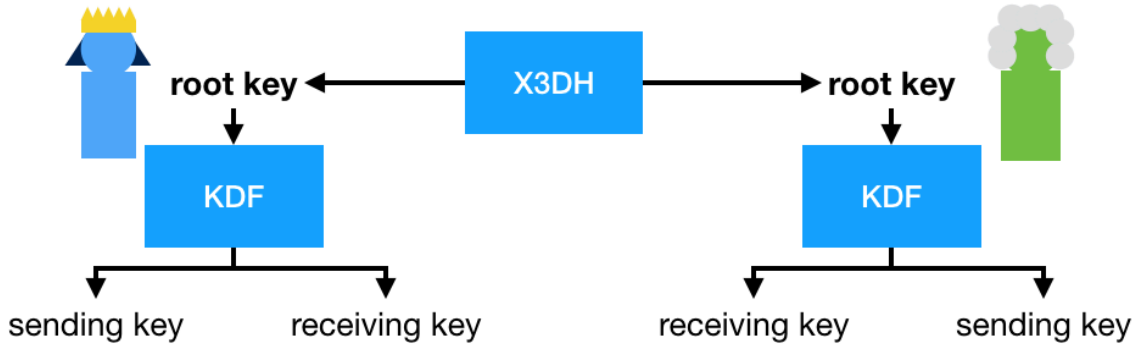


Figure 10.17 Building on figure 10.16, a better post-X3DH protocol would make use of a KDF with the output of the key exchanges to differentiate keys used to encrypt Bob's messages and to encrypt Alice's messages. Here Alice's sending key is the same as Bob's receiving key, and Bob's seending key is the same as Alice's receiving key.

This naive approach obviously does not provide any forward secrecy: if at one point in time this session key is stolen all previously recorded messages can be decrypted.

This post-X3DH protocol is better, but it still doesn't have forward secrecy. To fix this, Signal introduced what is called a **symmetric ratchet** as illustrated in figure 10.18. The sending key is now renamed a sending chain key, and not used directly to encrypt messages. When sending a message, Alice continuously passes that sending chain key into a one-way function that will produce the next sending chain key as well as the actual sending keys to encrypt her messages. (Bob on the other hand will have to do the same but with the receiving chain key.) Thus, by compromising one sending key or sending chain key, an attacker cannot recover previous keys. (And the same is true when receiving messages.)

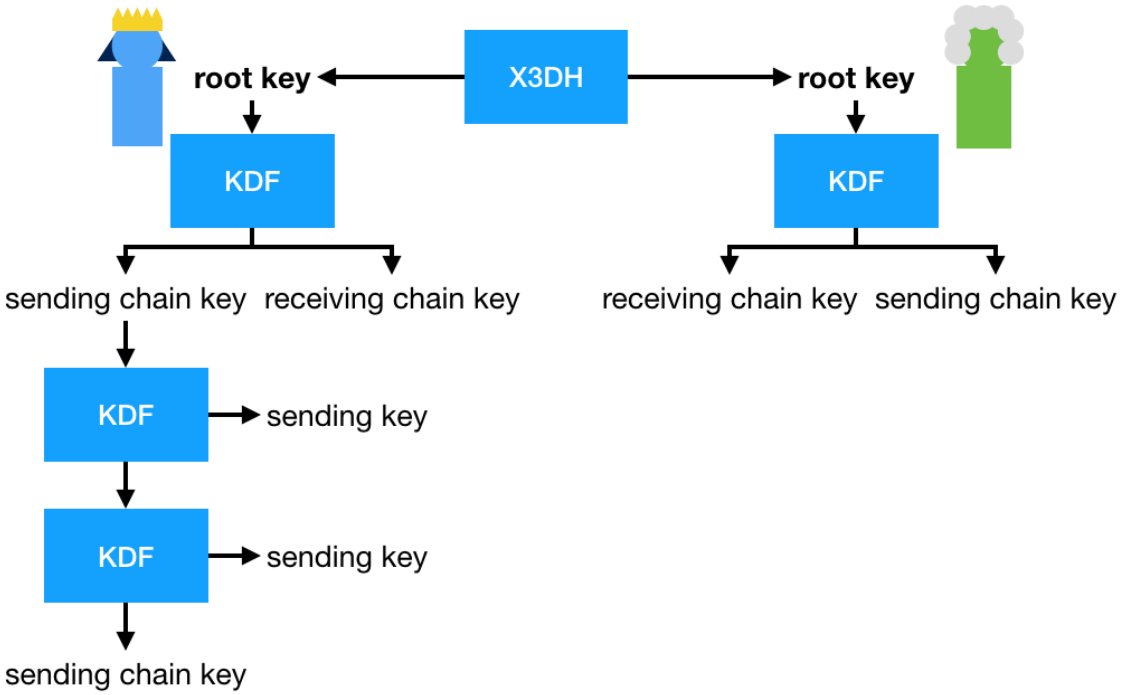


Figure 10.18 Building on figure 10.17, forward secrecy can be introduced in the post-X3DH protocol by "ratcheting" (passing into a KDF) a chain key every time one needs to send a message, and ratcheting another chain key every time one receives a message. Thus, the compromise of a sending or receiving chain key does not allow an attacker to recover previous ones.

Good. We now have forward secrecy baked into our protocol and at the message level. Every message sent and received protects all previously sent and received messages.

Note that this is somewhat debatable as an attacker who compromises a key probably does this by compromising a user's phone, which will likely contain all previous messages in clear next to the key. Nevertheless, if both users in a conversation decide to delete previous messages (for example by using Signal's "disappearing messages" feature) the forward secrecy property is achieved.

The Signal protocol has one last interesting thing I want to talk about: **post-compromise security** (also called backward secrecy as you've learned in chapter 8). Post-compromise security is the idea that if your keys get compromised at some point, you can still manage to recover from this as the protocol goes on and heals itself. Of course if the attacker still has access to your device after a compromise, then this is for nothing. Post-compromisee security can only work by reintroducing new entropy that a non-persistent compromise wouldn't have access to. The new entropy has to be the same for both peers. Signal's way of finding such entropy is by doing an ephemeral key exchange.

To do this, the Signal protocol continuously performs key exchanges in what is called a **Diffie-Hellman ratchet**. Every message sent by the protocol come with the current "ratchet

public key" as illustrated in figure 10.19.

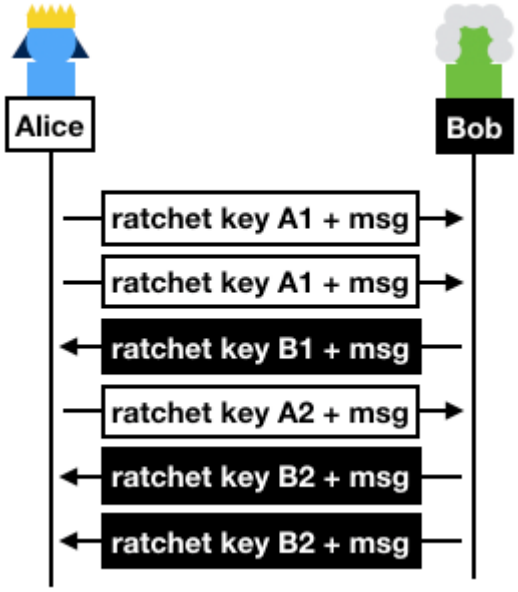


Figure 10.19 The Diffie-Hellman ratchet works by advertising a "ratchet public key" in every message sent. This ratchet public key can be the same as the previous one, or can advertise a new ratchet public key if a participant decides to refresh theirs.

When Bob notices a new ratchet key from Alice, he must perform a new Diffie-Hellman key exchange with Alice's new ratchet key and Bob's own ratchet key. The output can then be used with the symmetric ratchet to decrypt the messages received. I illustrate this in figure 10.20.

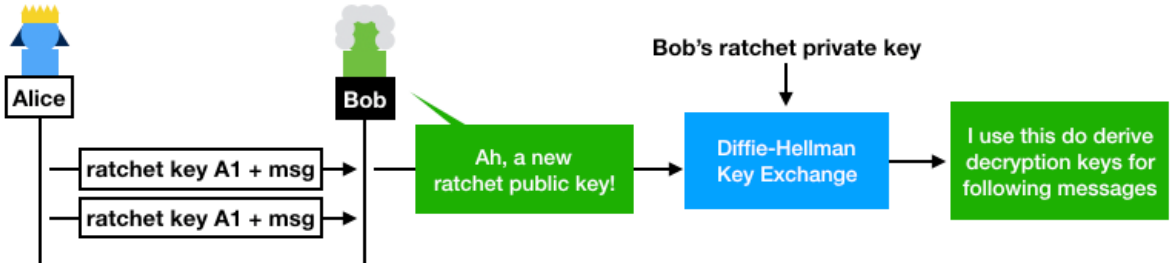


Figure 10.20 When receiving a new ratchet public key from Alice, Bob must do a key exchange with it and his own ratchet key to derive decryption keys (this is done with the symmetric ratchet). Alice's messages can then be decrypted.

Another thing that Bob must do when receiving a new ratchet key: generate a new random ratchet key for himself. With his new ratchet key, he can perform another key exchange with Alice's new ratchet key, which he will use to encrypt messages to her. This should look like figure 10.21.

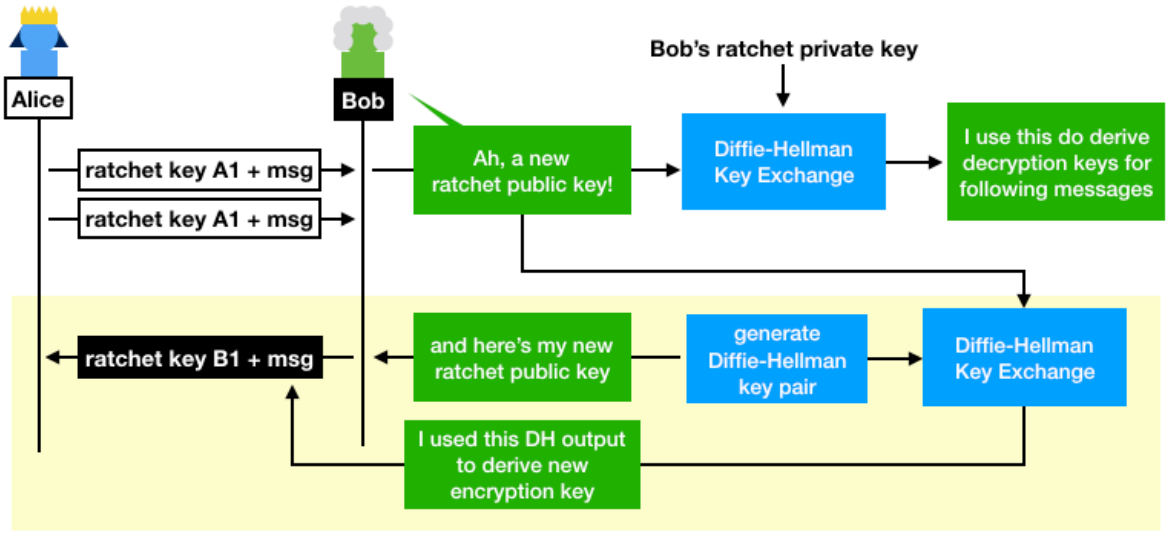


Figure 10.21 Building on figure 10.20, after receiving a new ratchet key Bob must also generate a new ratchet key for himself. This new ratchet key is used to derive encryption keys, and is advertised to Alice in his next series of messages (up until he receives a new ratchet key from Alice).

This back and forth of key exchanges is mentioned as a "ping pong" in the Double Ratchet specification:

> *This results in a "ping-pong" behavior as the parties take turns replacing ratchet key pairs. An eavesdropper who briefly compromises one of the parties might learn the value of a current ratchet private key, but that private key will eventually be replaced with an uncompromised one. At that point, the Diffie-Hellman calculation between ratchet key pairs will define a DH output unknown to the attacker.*
>
> *– The Double Ratchet Algorithm*

Finally, the combination of the Diffie-Hellman ratchet and the symmetric ratchet is called the **double ratchet**. It's a bit dense to visualize as one diagram, but figure XREF double_rachet attempts to do it.

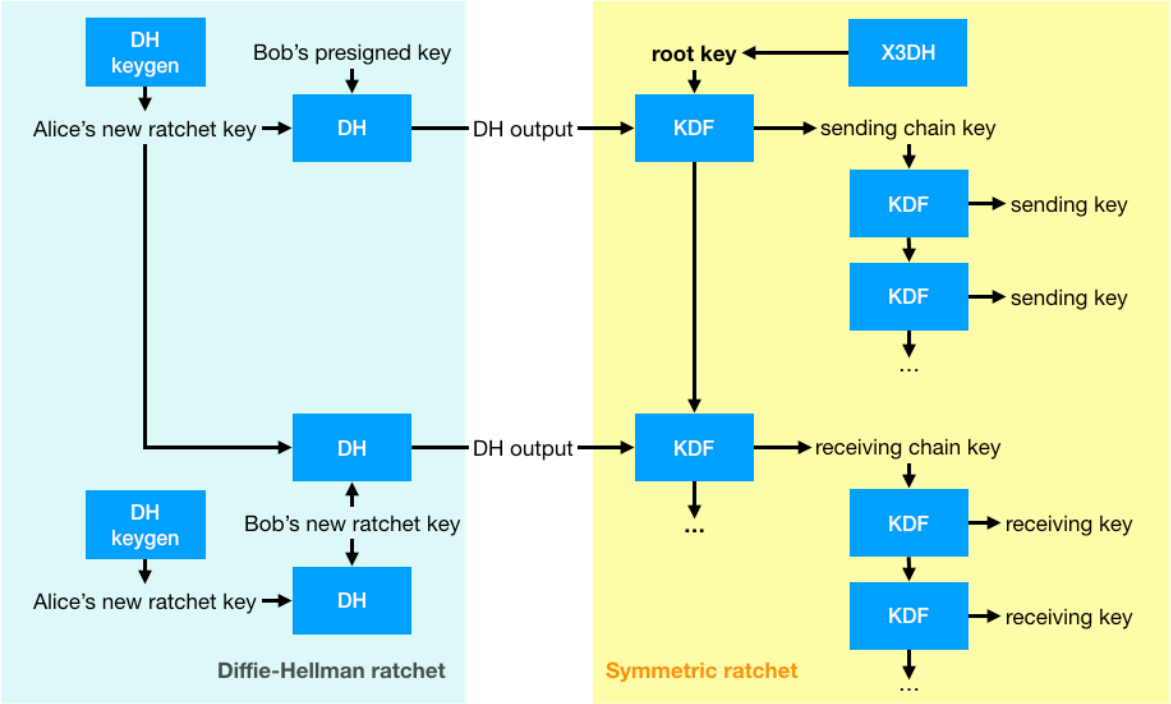


Figure 10.22 The double ratchet (from Alice's point of view) combines the Diffie-Hellman ratchet (on the left) with the symmetric ratchet (on the right), which provides post-compromise security as well as forward secrecy to the post-X3DH protocol. In the first message, Alice does not yet know Bob's ratchet key so she uses his presigned key instead.

I know this last diagram is quite dense, so I encourage you to take a look at Signal's specifications, which are published on https://signal.org/docs/ and which provide another well-written explanation of the protocol.

10.5 The state of end-to-end encryption

Today, most secure communications between users happen through secure messaging applications instead of encrypted emails. The Signal protocol has been the clear winner in its category, being adopted by many proprietary applications but also by open source and federated protocols like XMPP (via the OMEMO extension) and Matrix (a modern alternative to IRC). On the other hand, PGP and S/MIME are being dropped as published attacks have led to a loss of trust.

What if you want to write your own? Unfortunately a lot of what's being used in this field is ad-hoc, and you would have to fill many of the details yourself in order to obtain a full-featured and secure system.

Signal has open sourced a lot of its code, but it lacks documentation and can be hard to use correctly.[12]

On the other hand, you might have better luck using a decentralised and open source solution

like Matrix, which might prove easier to integrate with (and this is what the French government has done).[13] (See figure 10.23 for an illustration of a Matrix client.)

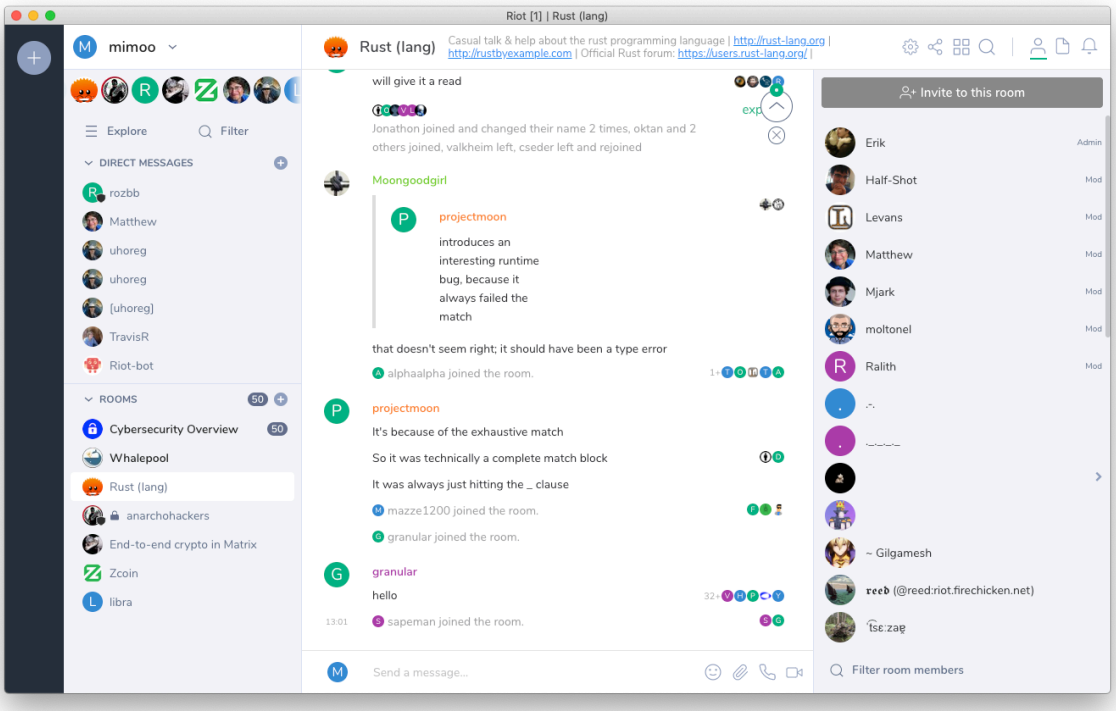


Figure 10.23 Riot, an end-to-end encrypted chat application using the Matrix network.

There are also a number of open questions and active research problems that I want to talk about:

- Group messaging.
- Support for multiple devices.
- Better security assurances than TOFU.

Let's start with the first item: **group messaging**. At this point, while implemented in different ways by different applications, group messaging is still being actively researched.

For example, the Signal application have clients make sense of group chats. Servers only see pairs of users talking, never less, never more. This means that clients have to encrypt a group chat message to all of the group chat participant, and send these individually. This is called **client-side fanout** and does not scale super well. It is also not too hard for the server to figure out what are the group members when it sees Alice sending several messages of the same length to Bob and Charles.

WhatsApp on the other hand use a variant of the Signal protocol where the server is aware of group chat membership.[14] This change allows a participant to send a single encrypted message to the server, which in turn will have the responsibility to forward it to the group members. This

is called **server-side fanout**.

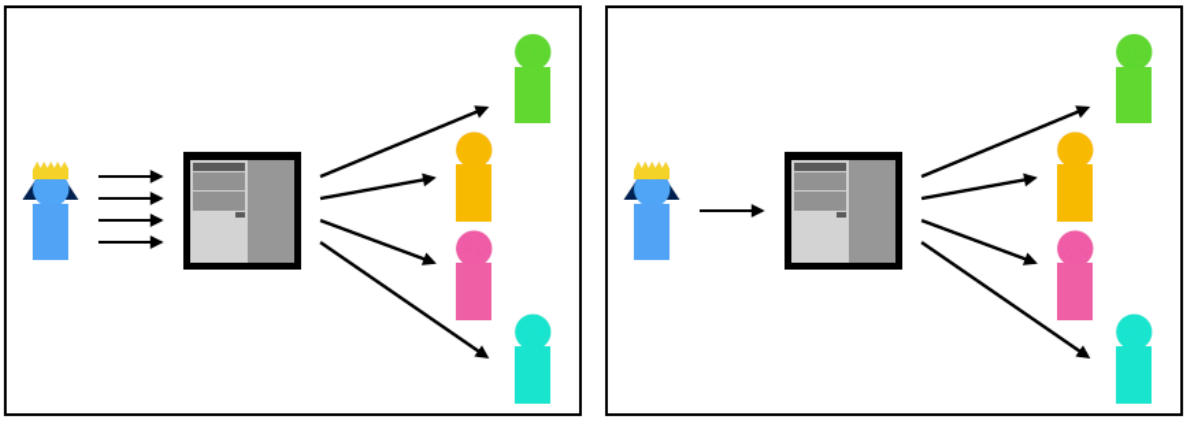


Figure 10.24 There are two ways to approach end-to-end encryption for group chats. A client-side fanout approach means that the client has to individually message each recipient usuing their already existing encrypted channel. This is a good approach to hide group membership from the server. A server-side fanout approach let the server forward a message to each group chat participant. This is a good approach to reduce the number of messages sent from the client perspective.

Another problem of group chat is scaling to groups of large memberset. For this, many players in the industry have recently gathered around a **Messaging Layer Security (MLS)** standard to tackle secure group messaging at scale, but there seems to be a lot of work to be done (and one can wonder, is there really any confidentiality in a group chat with more than a hundred participants?).

Note that this is still an area of active research, and different approaches come with different trade-offs in security and usability. In 2021, no group chat protocol seems to provide transcript consistency, which ensures that all participants see the same messages in the same order.

Support for multiple devices is either not a thing, or implemented in various way, most often by pretending that your different devices are different participants of a group chat. The TOFU model can make handling multiple devices quite complicated, as having different identity keys per device can become a real key management problem. Imagine having to verify fingerprints for each of your devices, and each of your friends' devices. For example, Matrix has a user signs their own devices. Other users then can trust all your devices as one entity by verifying their associated signatures.

Finally, I've mentioned that the TOFU model is also not the greatest, as it is based on trusting a public key the first time we see it, and most users do not verify later that fingerprints match. Can something be done about this? What if the server decides to impersonate Bob to Alice only? This is a problem that **Key Transparency** is trying to tackle.[15] Key Transparency is a protocol similar to the Certificate Transparency protocol that I've talked about in chapter 9. There is also some research making use of the blockchain technology that I'll talk about in chapter 12 on cryptocurrencies.

10.6 Summary

- End-to-end encryption is about securing communications betwen real human beings. A protocol implementing end-to-end encryption is more resilient to vulnerabilities that can happen in servers sitting in-between users, and can greatly simplify legal requirements for companies.
- End-to-end encryption systems need a way to bootstrap trust between users.This trust can come from a public key that we already know, or an out-of-band channel that we trust.
- PGP and S/MIME are the main protocols that are used to encrypt emails today, yet none of them are considered safe to use as they make use of old cryptographic algorithms and practices. They also have poor integration with email clients that have been shown to be vulnerable to different attacks in practice.
    - PGP uses a web of trust model where users sign each other public keys in order to allow others to trust them.
    - S/MIME uses a public key infrastructure to build trust between participants. It is most commonly used in companies and universities.
- An alternative to PGP today is saltpack, which fix a number of issues while relying on social networks to discover other people's public keys.
- Emails will always have issues with encryption, as the protocol was built without encryption in mind. On the other hand, modern messaging protocols and applications are considered better alternatives to encrypted emails, as they are built with end-to-end encryption in mind.
    - The Signal protocol is used by most messaging applications to secure end-to-end communications between users. For example Signal messenger, WhatsApp, Facebook Messenger, and Skype all advertise their use of the Signal protocol to secure messages.
    - Other protocols, like Matrix, attempt to standardize federated protocols for end-to-end encrypted messaging. Federated protocols are open protocols that anyone can interoperate with, as opposed to centralized protocols that are limited to a single application.

11 User authentication

This chapter covers

- User authentication based on passwords.
- User authentication based on symmetric and asymmetric keys.
- User-aided authentication and how humans can help secure connections between devices.

In the introduction of this book, I boiled cryptography down to two concepts: confidentiality and authentication. In real-world applications, confidentiality is (usually) the least of your problems, and authentication is where most of the complexity arises. I know I've already talked a lot about authentication throughout this book, but it can be a confusing concept as it is used with different meanings in cryptography. For this reason, this chapter starts with an introduction of what authentication really is about.

As usual with cryptography, no protocol is a panacea. The rest of the chapter will teach you about a number of authentication protocols that are being used in a multitude of real-world applications. So let's get started!

11.1 A recap on authentication

By now, you have heard of authentication many times, so let's recap. You've seen:

1. authentication in cryptographic primitives like message authentication codes (covered in chapter 3) and authenticated encryption (covered in chapter 4)
2. authentication in cryptographic protocols like TLS (covered in chapter 9) and Signal (covered in chapter 10) where one or both sides of the connection can be authenticated.

In the former case, authentication refers to the **authenticity** (or **integrity**) of messages. In the

latter case, authentication refers to **proving who you are to someone else**.

These are different concepts covered by the same word, which can be quite confusing! But both usages are correct, as the Oxford dictionary says:

> *Authentication. The process or action of proving or showing something to be true, genuine, or valid.*
>
> *– Oxford dictionary*

For this reason, you should think of authentication as a term used in cryptography to convey two different concepts depending on the context:

1. **Message/payload authentication**. You're proving that a message is genuine and hasn't been modified since its creation (for example, "are these messages authenticated or can someone tamper with them?")
2. **Origin/entity/identity authentication**. You're proving that an entity really is who they say they are (for example, "am I actually communicating with google.com?")

Bottom line: authentication is about proving that something is what it is supposed to be. And that thing can be a person, a message, or something else.

In this chapter, I will use the term authentication only to refer to identifying people or machines. In other words, identity authentication.

By the way, you've already seen a lot about this type of authentication:

- In chapter 9 on secure transport, you've learned that machines can authenticate other machines at scale by using public key infrastructures (so-called PKIs).
- In chapter 10 on end-to-end encryption, you've learned about ways humans can authenticate one another at scale by using trust on first use (TOFU) and fingerprints.

In this chapter, you will learn the two other cases not previously mentioned:

- **user authentication**, or how machines authenticate humans. Beep boop.
- **user-aided authentication**, or how humans can authenticate machines, or more accurately how humans can help their devices authenticate machines.

I recapitulate this in figure 11.1.

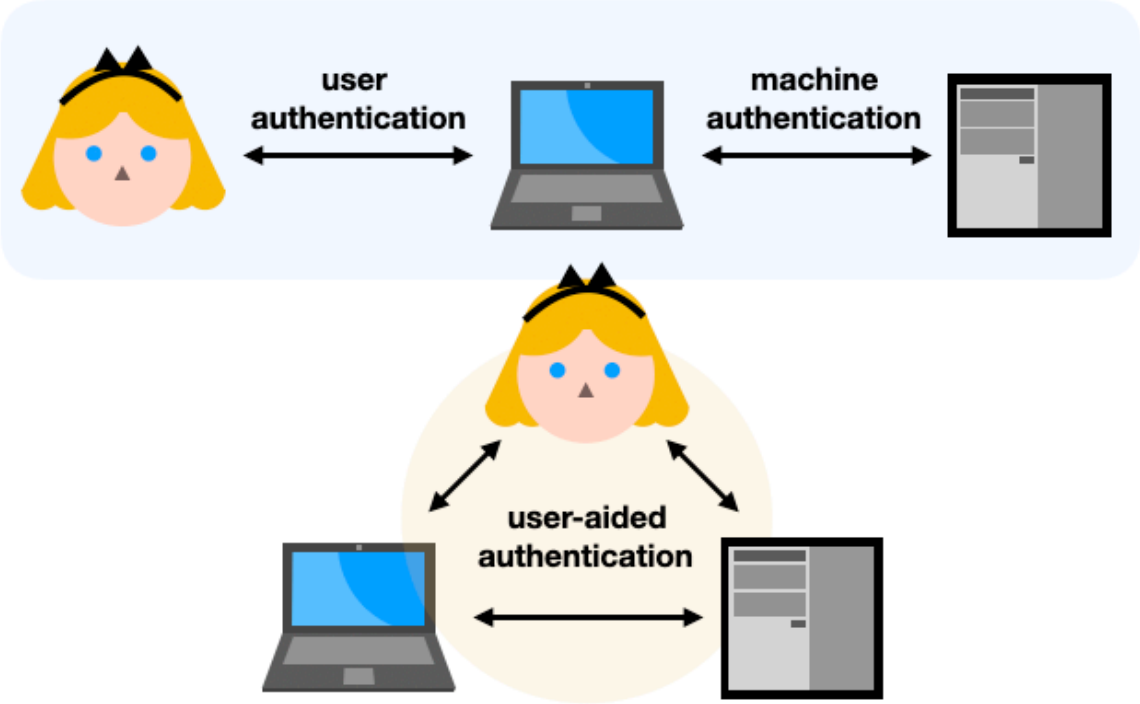


Figure 11.1 In this book I talk about origin authentication in three types of scenarios. User authentication happens when a device authenticates a human being. Machine authentication happens when a machine authenticates another machine. User-aider authentication happens when a human is involved in a machine authenticating another machine.

Another aspect of identity authentication is the identity part. Indeed, how do we define someone like Alice in a cryptographic protocol? How can a machine authenticate you and I? There is, unfortunately (or fortunately), an inherent gap between flesh and bits. To bridge reality and the digital world we always assume that, for example, Alice is the only one who knows some secret data. To prove her identity, Alice then has to demonstrate knowledge of that secret data. For example, she could be sending her password, or she could be signing a random challenge with the private key associated to her public key.

Alright, that's enough intro. If this section didn't make too much sense, the multitude of examples that are to follow will. Let's now first have a look at the many ways machines have found to authenticate us humans!

11.2 User authentication, or the quest to get rid of passwords

The first part of this chapter is about how machines authenticate humans, or in other words **user authentication**. There are many ways to do this, and no solution is a panacea. But in most user authentication scenarios, we assume that:

- the server is already authenticated
- the user shares a secure connection with it

For example, you can imagine that the server is authenticated to the user via the web public key infrastructure, and that the connection is secured via TLS (both covered in chapter 9). In a sense, most of this section is about **upgrading a one-way authenticated connection to a mutually-authenticated connection**, as illustrated in figure 11.2.

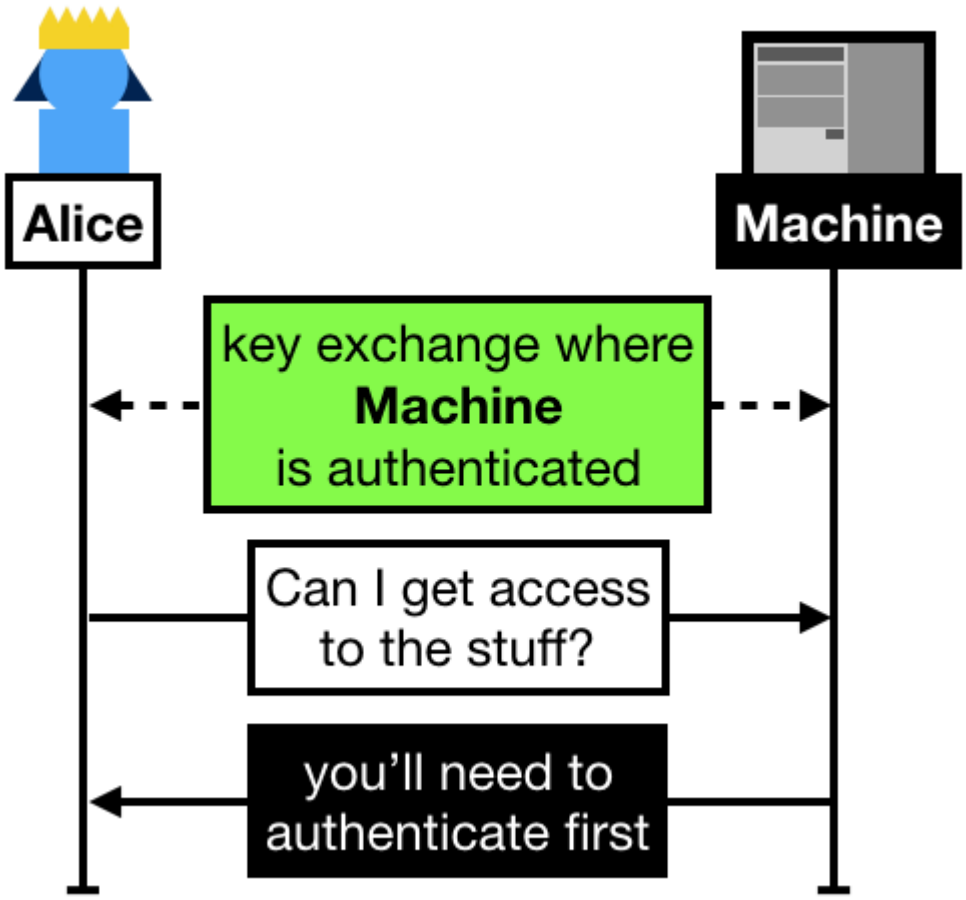


Figure 11.2 User authentication typically happens over a channel that is already secured, but where only the server is authenticated. A typical example is when you browse the web using HTTPS and log into a webpage using your credentials.

I have to warn you, user authentication is a vast land of broken promises. You must have used **passwords** many times to authenticate to different webpages, and your own experience probably resembles something like this:

- You register with a username and password on a website.
- You log into the website using your new credentials.
- You change your password after recovering your account or because the website forces you to.
- If you're out of luck, your password (or a hash of it) is leaked in a series of database breaches.

Sounds familiar?

> **NOTE** I will ignore password/account recovery in this chapter, as they have little to do with cryptography. Just know that they are often tied to the way you first registered (for example, if you registered with the IT department of your workplace, then you'll probably have to go see them if you lose your password) and they can be the weakest link in your system if you are not careful. Indeed, if I can recover your account by calling a number and giving someone your birth date, then no amount of cool cryptography at login time will help.

A naive way to implement the previous user authentication flow is to store the user password at registration, and then ask the user for it at login time (as illustrated in figure 11.3). As covered in chapter 3, once successfully authenticated a user is typically given a cookie that she can send in every subsequent request instead of her username and password.

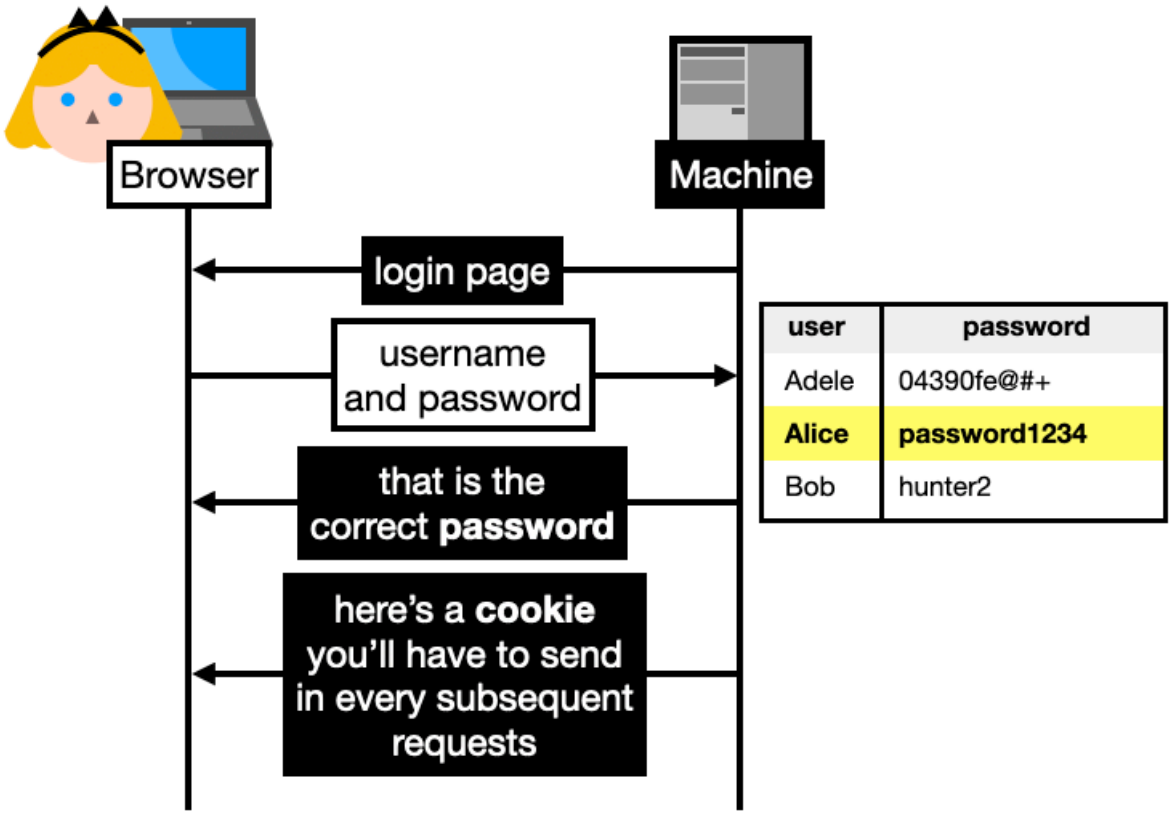


Figure 11.3 A password-based authentication flow. Alice sends her username and password in clear to the server (the server sees the password), and the server checks if it matches what it has in its database. If it does, Alice successfully authenticates.

But wait, if the server stores your password in clear, then any breach of their databases will reveal your password to the attackers. These attackers will then be able to use it to log into any websites where you used the same password to register. A better way to store passwords would be to use a **password hashing** algorithm like the standardized **Argon2** you've learned about in

chapter 2. This would effectively prevent the server to see your password every time you log in (although they would still see it at registration time). Yet, a lot of websites and companies still store passwords in clear.

> **NOTE** **Sometimes, login applications will attempt to fix the issue of the server learning about their passwords at registration time by having the client hash (or password hash) the password before sending it to the server. Can you tell if this really works?**

Moreover, humans are naturally bad at passwords. We are usually most comfortable with small and easy-to-remember passwords. And if possible, we would want to just reuse the same password everywhere.

81% of all hacking-related breaches leverage stolen or weak passwords.
– Verizon Data Breach Report (2017)

The problem of weak passwords and password reuse has led to many silly and annoying design patterns that attempt to force users to take passwords more seriously. For example, some websites will require you to use special characters in your passwords, or will force you to change password every 6 months, and so on.

Furthermore, many protocols attempt to "fix" passwords or to get rid of them altogether. Every year, new security experts seem to think that the concept of password is dead. Yet, it is still the most widely used user authentication mechanism.

So here you have it, passwords are probably here to stay. Yet, there exist many protocols that improve or replace passwords. Let's take a look at them!

11.2.1 One password to rule them all, single sign-on (SSO) and password managers

OK, password reuse is bad, what can we do about it? Naively, users could use different passwords for different websites, but there are two problems with this approach:

- Users are bad at creating many different passwords.
- The mental load required to remember multiple passwords is impractical.

To alleviate these concerns, two solutions have been widely adopted:

- **Single-sign on (SSO)**. The idea of SSO is to allow users to connect to many different services by proving that they own the account of a single service. This way the user only has to remember the password associated with that one service in order to be able to connect to many services. Think "connect with Facebook" type of buttons, as illustrated in figure 11.4.
- **Password Managers**. The previous SSO approach is convenient if the different services you use all support it, but this is obviously not scalable for scenarios like the web. A better approach in these extreme cases is to improve the client as opposed to attempting to fix the issue on the (too many) servers' side. Nowadays, modern browsers have built-in password managers that can suggest complex passwords when you register on new websites, and can remember all of these passwords as long as you remember one master password.

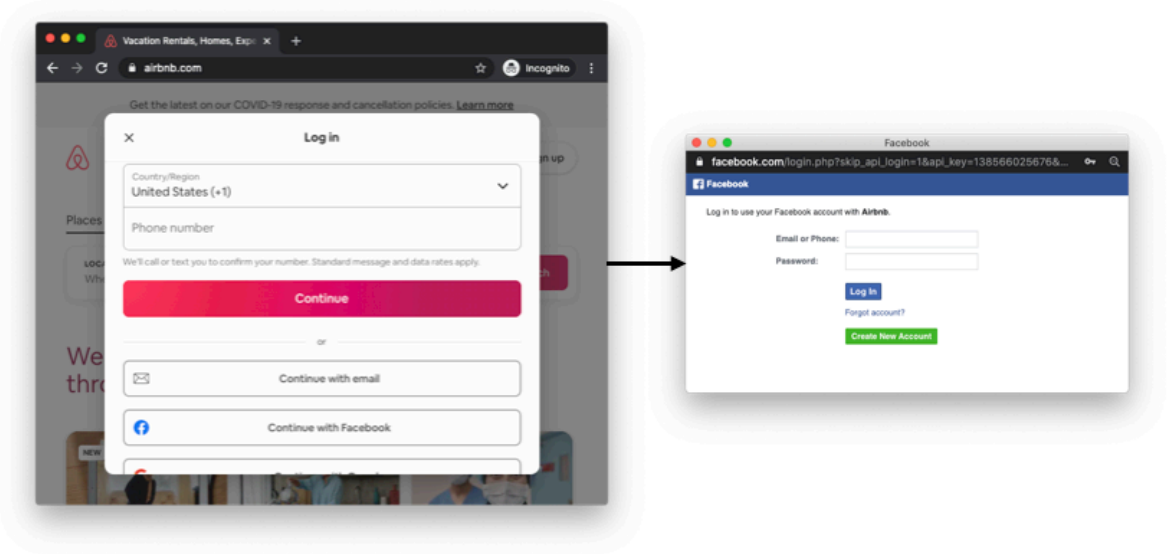


Figure 11.4 An example of single-sign on (SSO) on the web. By having an account on Facebook or Google, a user can connect to new services (in this example Airbnb) without having to think about a new password.

The concept of SSO is not new in the enterprise world, but its success with normal end-users is relatively recent. Today, two protocols are the main competitors when it comes to setting up SSO:

- **Security Assertion Markup Language 2.0 (SAML)**. A protocol using the Extensible Markup Language (XML) encoding.
- **OpenID Connect (OIDC)**. An extension to the **OAuth 2.0** (RFC 6749) authorization protocol using the JavaScript Object Notation (JSON) encoding.

SAML is still widely used, mostly in an enterprise setting, but it is at this point a legacy protocol.

OpenID Connect, on the other hand, can be seen everywhere on web and mobile applications. You most likely already used it!

While OpenID Connect allows for different types of flows, let's see the most common use case for user authentication on the web via the **authorization code flow**:

1. Alice wants to log into some application, let's say `example.com`, via an account she owns on `cryptologie.net` (that's just my blog, but let's pretend that you can register an account on it).
2. `example.com` redirects her browser to a specific page of `cryptologie.net` to request an "authorization code." If she is not logged-in in `cryptologie.net`, the website will first ask her to log in. If she is already logged-in, the website will still confirm with the user that they want to connect to `example.com` using their identity on `cryptologie.net` (it is important to confirm user intent).
3. `cryptologie.net` redirects Alice back to `example.com` which then learns the authorization code.
4. `example.com` can then query `cryptologie.net` with this authorization code to confirm Alice's claim that she owns an account on `cryptologie.net`, and potentially retrieve some additional profile information about that user.

I recapitulate this flow in figure 11.5.

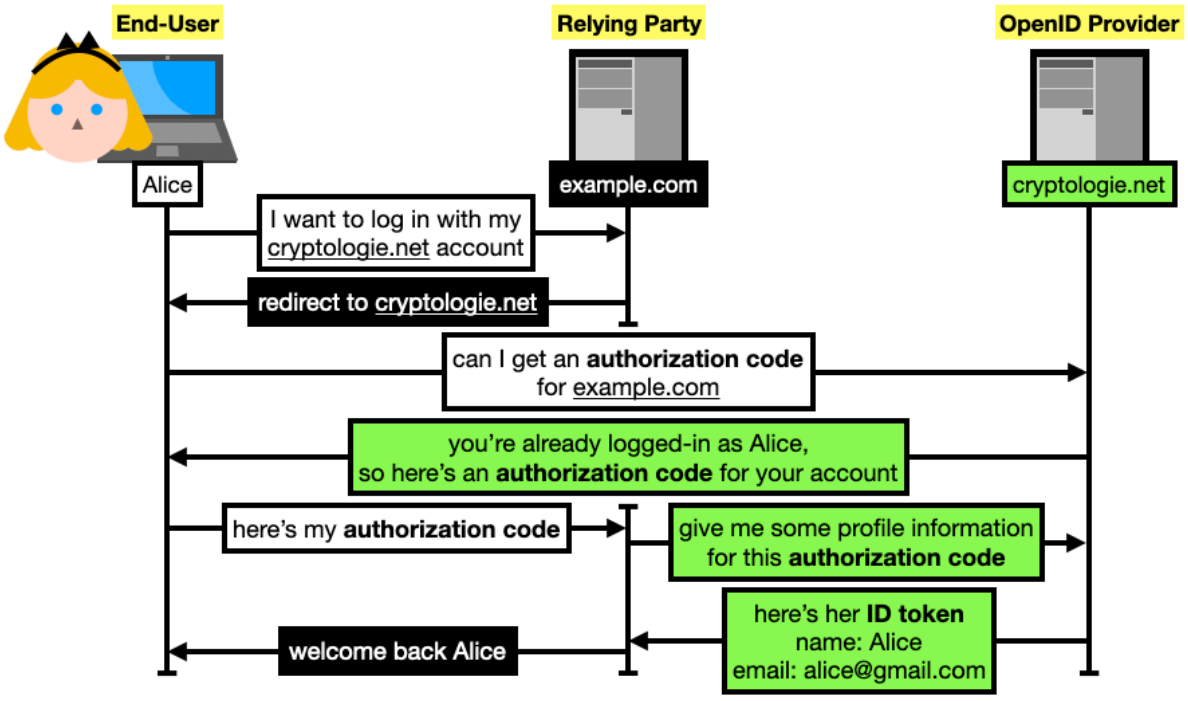


Figure 11.5 In OpenID Connect (OIDC), Alice (the end-user in OIDC terms) can authenticate to a service `example.com` (the relying party) using her already existing account on `cryptologie.net` (the OpenID provider). For the web, the authorization code flow of OIDC is usually used. It starts by having Alice request an "authorization code" from `cryptologie.net` (and that can only be used by `example.com`). `example.com` can then use it to query `cryptologie.net` for Alice's profile information (encoded as an ID token), and then associate her `cryptologie.net` identity with an account on their own platform.

There are many important details that I am omitting here. For example, the authorization token that Alice receives in step 2 must be kept secret, as it can be used to log in as her on `example.com`. Another example: so that `example.com` cannot reuse the authorization token to connect as Alice on a different website, the OpenID provider `cryptologie.net` retains an association between this authorization token and the intended audience (`example.com`). This can be done by simply storing this association in a database, or by having the authorization code contain this information authenticated (I explained a similar technique with cookies in chapter 3).

By the way, the proof given to `example.com` in step 4 is called an **ID token** in OpenID Connect, and is represented as a **JSON Web Token (JWT)** which is just a list of JSON-encoded data. I illustrate JWTs in figure 11.6.

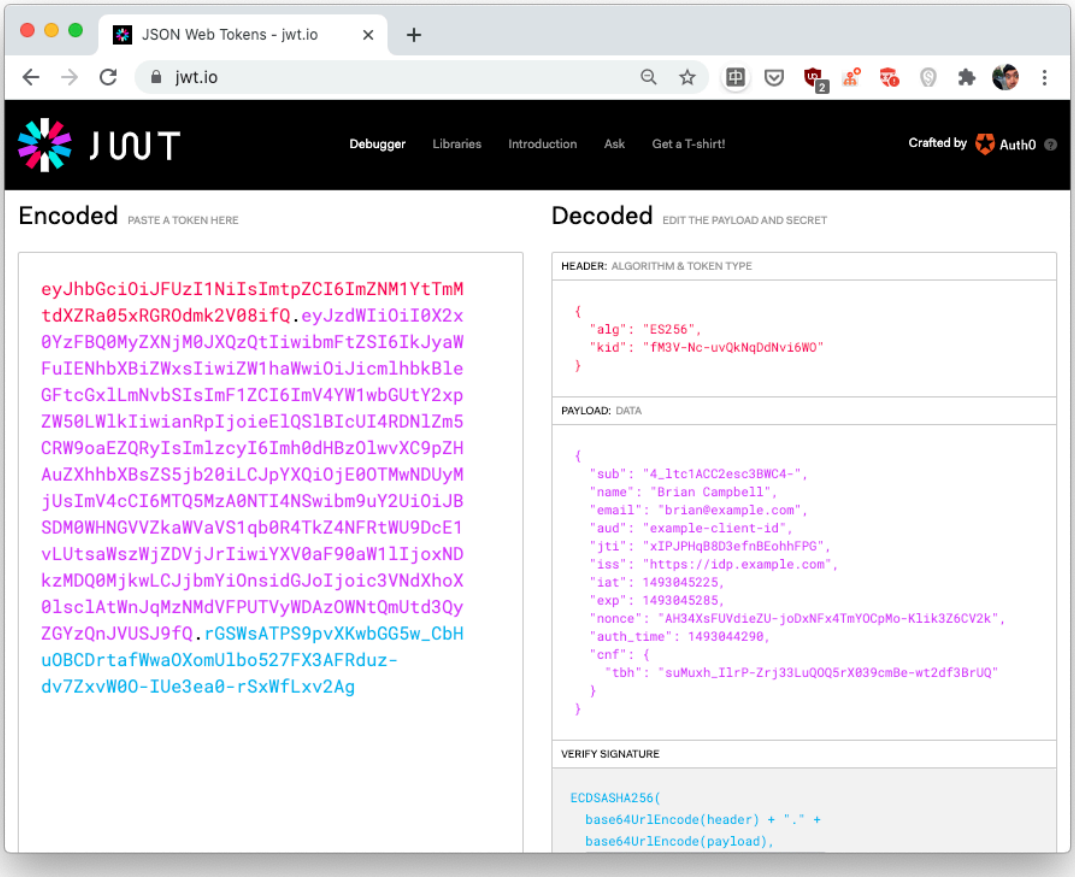


Figure 11.6 In OpenID Connect, if Alice wants to log in on `example.com` via her account on another website (an OpenID provider), `example.com` eventually needs to obtain what is called an "ID token." An ID token is some user information encoded via the JSON Web Token (JWT) standard. a JWT is simply the concatenation of three JSON-encoded objects. In the picture, the website https://jwt.io lets you decode a JWT and learn what every field stands for. In our browser-based example, `example.com` uses TLS to communicate with (and authenticate) the OpenID provider. Thus, the ID token it receives can be trusted. In other OpenID Connect flows (used, for example, by mobile applications) the ID token can be provided directly by the user, and can thus be tampered with. In this case, an ID token contains a signature which can be verified using the OpenID provider's public key.

Authentication protocols are often considered hard to get right. OAuth2, the protocol OpenID Connect relies on, is notorious for being easy to mis-use.[16] On the other hand, OpenID Connect is very well specified.[17] Make sure that you follow the standards and that you look at best practices, this can save you from a lot of trouble.

> **NOTE** Here's another example of a pretty large company deciding not to follow this advice. In May 2020, the "Sign-in with Apple" SSO flow that took a departure from OpenID Connect was found to be vulnerable. Anyone could have obtained a valid ID token for any Apple account, just by querying Apple's servers.[18]

SSO is great for users, as it reduces the number of passwords they have to manage, but it does not remove passwords altogether. The user still has to use passwords to connect to OpenID providers. So next, let's see how cryptography can help hide passwords!

11.2.2 Don't want to see their passwords? Use an asymmetric password-authenticated key exchange

The previous section surveyed solutions that attempt at simplifying identity management for users, allowing them to authenticate to multiple services using only one account linked to a single service. While protocols like OpenID Connect are great, as they effectively decrease the number of passwords users have to manage, they don't change the fact that some service still needs to see the password of the user in clear. (Even if the password is stored after password hashing it, it is still sent in clear every time the user registers, changes their password, or logs in.)

Cryptographic protocols called **asymmetric (or augmented) password-authenticated key exchanges (PAKEs)** attempt at providing user authentication without having the user ever communicate their passwords directly to the server. They contrast with **symmetric or balanced PAKEs** protocols where both sides know the password.

The most popular asymmetric PAKE at the moment is the **Secure Remote Password (SRP)** protocol, which was standardized for the first time in 2000,[19] and later integrated into TLS.[20]

It is quite an old protocol, and has a number of flaws.[21] For example, if the registration flow is intercepted by a MITM attacker, the attacker would then be able to impersonate and log in as the victim. It also does not play well with modern protocols, it cannot be instantiated on elliptic curves and worse, it is incompatible with TLS 1.3.

Since then, many asymmetric PAKEs have been proposed and standardized. In the summer of 2019, the Crypto Forum Research Group (CFRG) of the IETF started a PAKE selection process,[22] with the goal to pick one algorithm to standardize for each category of PAKEs

(symmetric/balanced and asymmetric/augmented).

In march 2020, the CFRG announced the end of the PAKE selection process, selecting:

- **CPace** as the recommended symmetric/balanced PAKE (invented by from Björn Haase and Benoît Labrique)
- **OPAQUE** as the recommended asymmetric/augmented PAKE (invented by Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu)

In this section I will talk about OPAQUE, which in early 2021 is still in the process of being standardized. In the second section of this chapter, you will learn more about symmetric PAKEs and CPace.

OPAQUE takes its name from the homonym O-PAKE, where O refers to the term "oblivious." This is because OPAQUE relies on a cryptographic primitive that I have not yet mentioned in this book: an **Oblivious Pseudo-Random Function (OPRF)**.

OBLIVIOUS PSEUDO-RANDOM FUNCTIONS (OPRFS)

OPRFs are a two-participant protocol that mimics the PRFs that you've learned about in chapter 3. As a reminder, a PRF is somewhat equivalent to what one would expect of a MAC: it takes a key and an input and gives you a totally random output of a fixed length.

> **NOTE** **The term oblivious in cryptography generally refers to protocols where one party computes a cryptographic operation without knowing the input provided by another party.**

Here is how an OPRF works at a high level:

1. Alice wants to compute a PRF over an input, but wants the input to remain secret. She "blinds" her input with a random value called a "blinding factor" and sends this to Bob.
2. Bob runs the OPRF on this blinded value with his secret key, the output is still blinded so useless for Bob. Bob then sends this back to Alice.
3. Alice finally "unblinds" the result, using the same blinding factor she had previously used, to obtain the real output.

It is important to note that every time Alice wants to go over this protocol, she has to create a different blinding factor. But no matter what blinding factor she uses, as long as she uses the same input she will always obtain the same result. I illustrate this in figure 11.7.

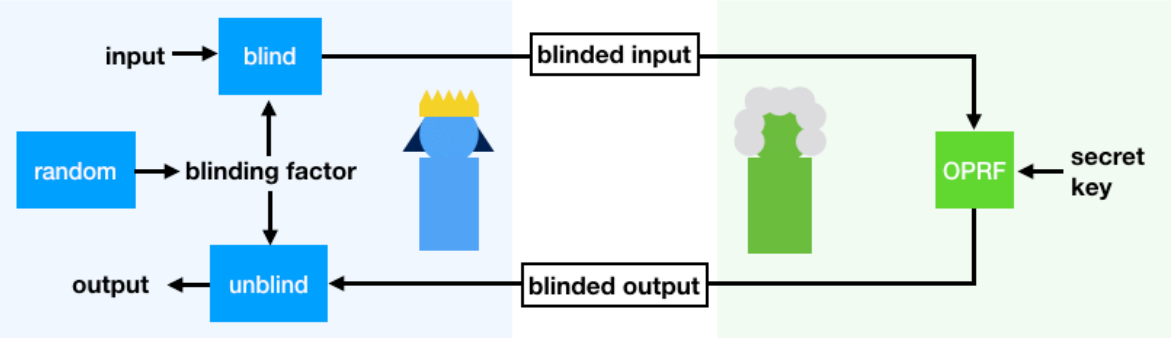


Figure 11.7 An oblivious PRF (OPRF) is a construction that allows one party to compute a PRF over the input of another party without learning that input. To do this, Alice first generates a random blinding factor, then blinds her input with it before sending it to Bob. Bob uses his secret key to compute the PRF over the blinded value, then send the blinded output to Alice who can then unblind it. The result does not depend on the value of the blinding factor.

Here's an example of an OPRF protocol implemented in a group where the discrete logarithm problem is hard:

1. Alice converts her input to a group element x.
2. Alice generates a random blinding factor r.
3. Alice blinds her input by computing blinded_input = x^r.
4. Alice sends the blinded_input to Bob.
5. Bob computes blinded_output = blinded_input^k where k is the secret key.
6. Bob sends the result back to Alice.
7. Alice can then unblind the produced result by computing output = blinded_output$^{1/r}$ = x^k where 1/r is the inverse of r.

How OPAQUE uses this interesting construction is the whole trick behind the asymmetric PAKE!

THE OPAQUE ASYMMETRIC PAKE, HOW DOES IT WORK?

The idea is that we want a client, let's say Alice, to be able to do an authenticated key exchange with some server. We also assume that Alice already knows the server's public key, or already has a way to authenticate it (the server could be an HTTPS website and thus Alice can use the web PKI).

Let's see how we could build this, to progressively understand how OPAQUE works.

First idea: use public-key cryptography to authenticate Alice's side of the connection. If Alice owns a **long-term keypair** (and the server knows the public key) she can simply use it to perform a mutually-authenticated key exchange, or sign a challenge given by the server.

Unfortunately, an asymmetric private key is just too long and Alice can only remember her password… She could store a keypair on her current device, but she also want to be able to log in

from another device later.

Second idea: Alice can derive the asymmetric private key from her password, using the password-based key derivation functions (like Argon2) that you've learned about in chapter 2 and chapter 8. Alice's public key could then be stored on the server. If we want to avoid someone testing a password against the whole database (in the case of a database breach) we can have the server supply each user with a different salt that they have to use with the password-based key derivation function.

This is pretty good already, but there's one attack that OPAQUE wants to discard: **pre-computation attacks**. I can try to log in as you, receive your salt, and then pre-compute a huge number of asymmetric private keys (and their associated public keys) **offline**. The day the database is compromised, I can quickly see if I can find your public key (and the associated password) in my huge list of pre-computed asymmetric public keys.

Third idea: That's where the **main trick of OPAQUE** comes in! We can use the **OPRF protocol with Alice's password** in order to derive the asymmetric private key. If the server uses a different key per-user, that's as good as having salts (attacks can only target one user at a time). This way, an attacker that wants to pre-compute asymmetric private keys, based on guesses of our password, has to perform **online** queries. Online queries are good, because they can be rate-limited in order to prevent these kinds of online brute-force attacks.

> NOTE
>
> OPAQUE goes a bit further. Instead of having the user derive an asymmetric private key, it has the user derive a symmetric key. The symmetric key is then used to encrypt a backup of your asymmetric keypair and some additional data (like the server's public key). In this case, it prevents one more attack: offline brute-force attacks. Indeed, since you have to retrieve a backup of your keypair as part of the login flow, an attacker can use the backup to test password guesses offline (by trying to decrypt the encrypted backup with each symmetric key derived from password guesses). Due to the use of an OPRF, OPAQUE forces every password guess to be performed online.

Pretty clever no? I illustrate this technique in figure 11.8.

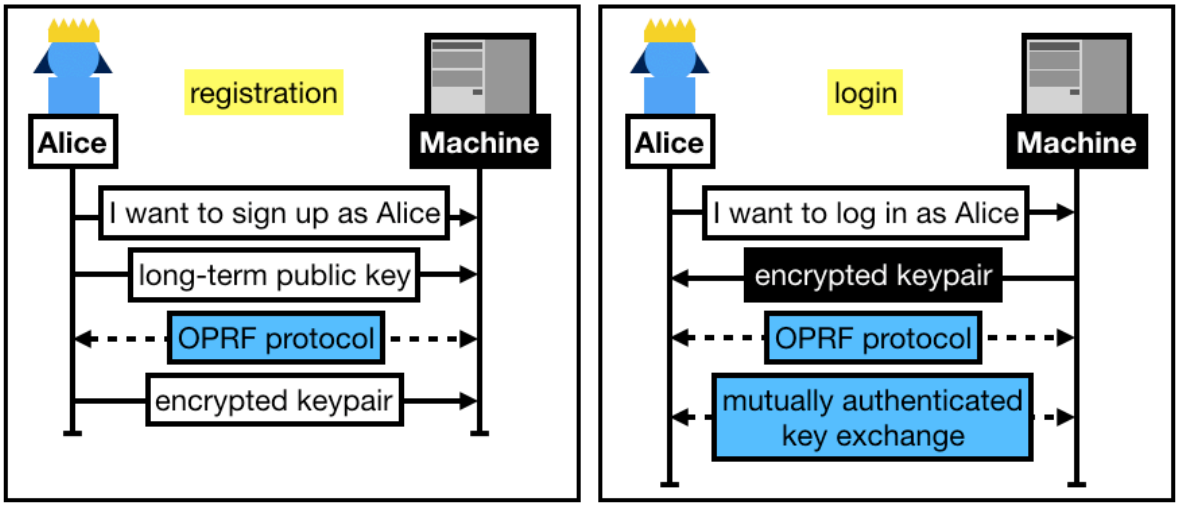


Figure 11.8 To register to a server using OPAQUE, Alice generates a long-term keypair and send her public key to the server who stores it and associates it with Alice's identity. She then uses the OPRF protocol to obtain a strong symmetric key from her password, and send an encrypted backup of her keypair to the server. To log in, she obtains her encrypted keypair from the server, then performs the OPRF protocol with her password to obtain a symmetric key capable of decrypting her keypair. All that's left is to perform a mutually-authenticated key exchange (or possibly sign a challenge) with this key.

Before going to the next section, let's recapitulate what you've learned here in figure 11.9.

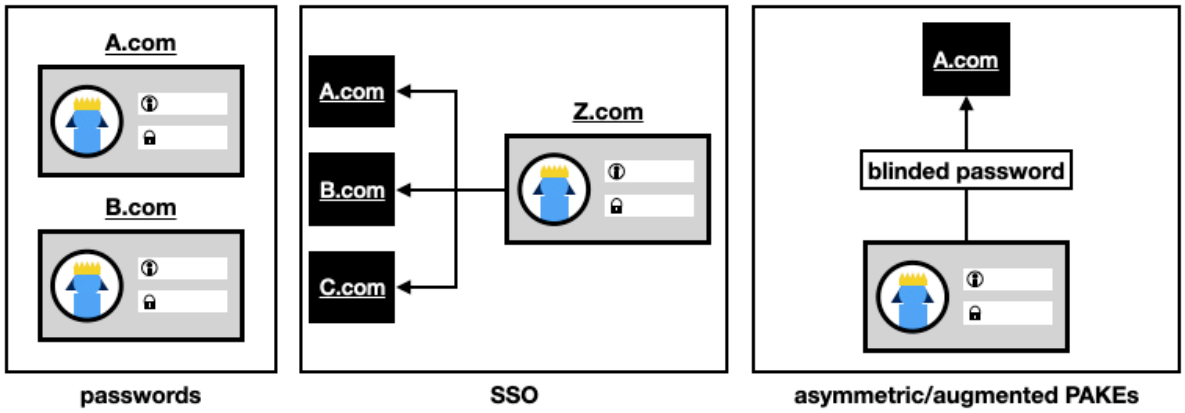


Figure 11.9 Passwords are a very handy way to authenticate users, as they live in someone's head and can be used on any devices. But on the other hand, users have trouble creating many strong passwords, and password breaches can be damaging as users tend to reuse passwords across websites. Single-sign on (SSO) allows you to connect to many services using one (or a few) service(s), while asymmetric (or augmented) password-authenticated key exchanges allow you to authenticate without the server ever learning your real password.

11.2.3 One-time passwords aren't really passwords, going passwordless with symmetric keys

Alright, so far so good. You've learned about different protocols that applications can leverage to authenticate users with passwords. But as you've heard, passwords are also not that great, they are vulnerable to brute-force attacks, tend to be reused, stolen, and so on. So what is available to us if we can afford to avoid passwords?

Keys!

And as you know there are two types of keys in cryptography, and both types can be useful:

- Symmetric keys.
- Asymmetric keys.

This section goes over solutions that are based on symmetric keys, while the next section will go over solutions based on asymmetric keys.

Let's imagine that Alice registers with a service using a symmetric key (often generated by the server and communicated to you via a QR code). A naive way to authenticate Alice later would be to simply ask her to send the symmetric key. This is of course not great, as a compromise of her secret would give an attacker unlimited access to her account. Instead, Alice can derive what are called **one-time passwords (OTPs)** from the symmetric key and send that in place of the longer-term symmetric key. Even though an OTP is not a password, the naming indicates that an OTP can be used in place of a password, and warns that it should never be reused.

The idea behind OTP-based user authentication is straightforward: your security comes from the knowledge of a (usually 16 to 32-byte uniformly random) symmetric key instead of a low-entropy password. This symmetric key allows you to generate one-time passwords on demand, as illustrated by figure 11.10.

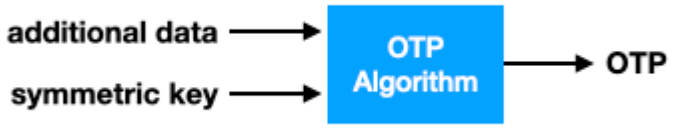


Figure 11.10 A one-time password (OTP) algorithm allows you to create as many one-time passwords as you want from a symmetric key and some additional data. The additional data is different depending on the OTP algorithm.

There are two main schemes that one can use to produce OTPs:

- The **HMAC-based One-time Password algorithm (HOTP)** standardized in RFC 4226. An OTP algorithm where the additional data is a counter.
- The **Time-based One-Time Password algorithm (TOTP)** standardized in RFC 6238.

An OTP algorithm where the additional data is the time.

Most applications nowadays use TOTP, as HOTP requires you to store a state (a counter) on both sides which might lead to issues if states fall out of synchrony.

OTP-based authentication is most often implemented in mobile applications (see figure 11.11 for a popular example) or in security keys (a small device that you can plug in the USB port of your computer).

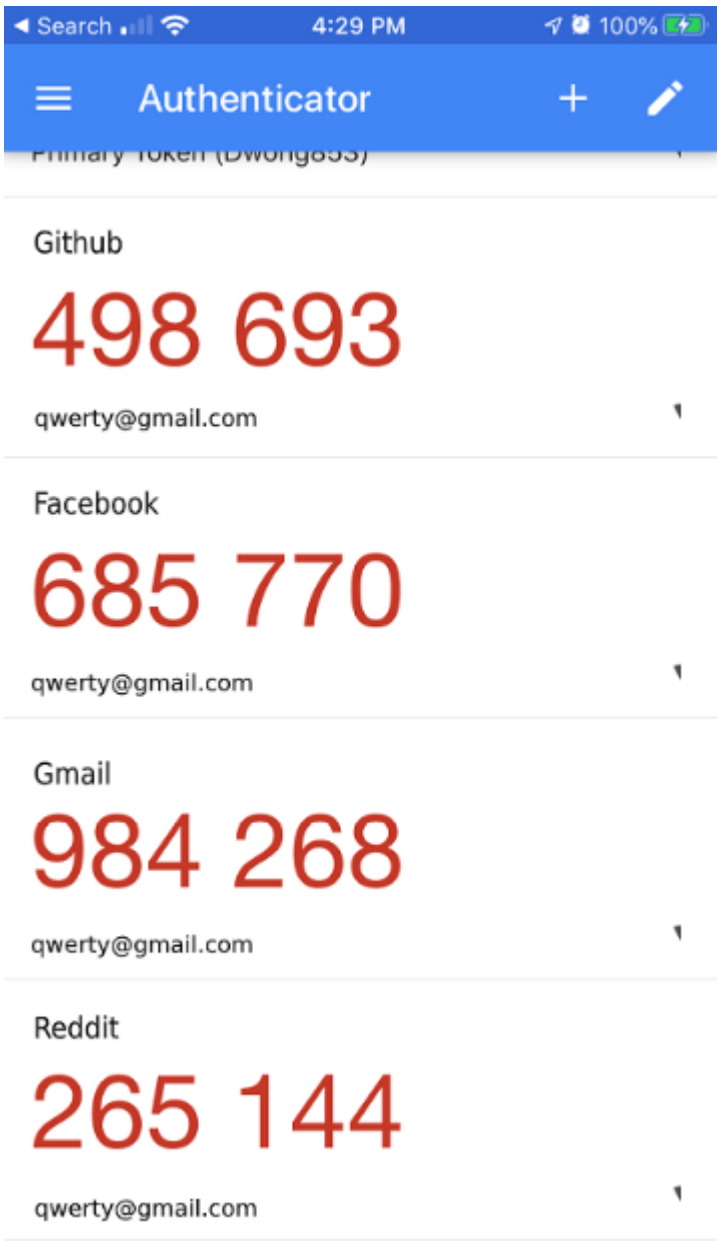


Figure 11.11 A screenshot of the Google Authenticator mobile app. The application allows you to store unique per-application symmetric keys, which can then be used with the TOTP scheme to generate 6-digit one-time passwords (OTPs) valid only for 30 seconds.

In most cases, this is how TOTP works:

At registration time, the service communicates a symmetric key to the user (perhaps using a QR code). The user then adds this key to a TOTP application.

At login time, The user can use the TOTP application to compute a one-time password. This is done by computing `HMAC(symmetric_key, time)` where `time` represents the current time (rounded to the minute in order to make a one-time password valid for 60 seconds). The TOTP application then displays the derived one-time password truncated and in a human-readable base to the user (for example reduced to 6 characters in base 10 to make it all digits). The user then either copies or types the one-time password into the relevant application. The application retrieves the user's associated symmetric key, and computes the one-time password in the same way as the user did. If the result matches the received one-time password, the user is successfully authenticated.

I recapitulate this flow in figure 11.12.

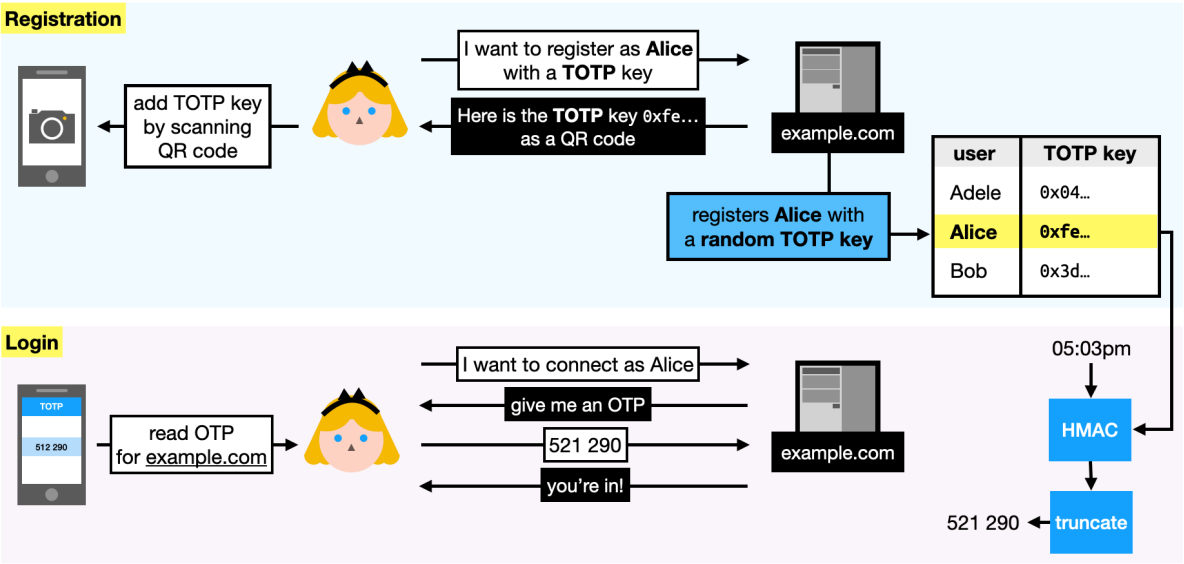


Figure 11.12 Alice registers with example.com using TOTP as authentication, by importing a symmetric key from the website into her TOTP application. Later, she can ask the application to compute a one-time password for example.com, and use it to authenticate with the website. Example.com just has to fetch the symmetric key associated to Alice, and compute the one-time password as well (using HMAC and the current time), then compare it in constant-time with what she sent.

Of course, similar to MAC authentication tag checks, the comparison between the user's OTP and the one computed on the server must be done in constant-time.

This authentication flow is not ideal though. There are a number of things that could be improved, for example:

- The authentication can be faked by the server, as they also own the symmetric key.
- You can be social engineered out of your one-time password.

> **NOTE** Phishing (or social engineering) is an attack that does not target vulnerabilities in the software, but rather vulnerabilities in human beings. Imagine that an application requires you to enter a one-time password to authenticate. What an attacker could do in this case, is to attempt to log in the application as you, and when prompted with a one-time password request, give you a call to ask you for a valid one (pretending that they work for the application). You're telling me you wouldn't fall for it, but good social engineers are very good at spinning believable stories and fabricating a sense of urgency that would make the best of us spill the beans. If you think about it, all the protocols that we've talked about previously are vulnerable to these types of attack.

For this reason, symmetric keys are yet another not-perfect replacement for passwords.

Next, let's see how using asymmetric keys can address these downsides.

11.2.4 Replacing passwords with asymmetric keys

Now that we're dealing with public-key cryptography, there's more than one way we can use asymmetric keys to authenticate to a server:

1. by using our asymmetric key inside a key exchange to authenticate our side of the connection.
2. by using our asymmetric key in an already-secured connection with an authenticated server.

Let's see each method.

MUTUAL AUTHENTICATION IN KEY EXCHANGES

You've already heard about the first method. In chapter 9, I mentioned that a TLS server can request the client to use a **certificate** as part of the handshake. Often, companies will provision their employees' devices with a unique per-employee certificate that will allow them to authenticate to internal services. See figure 11.13 for an idea of what it looks like from a user perspective.

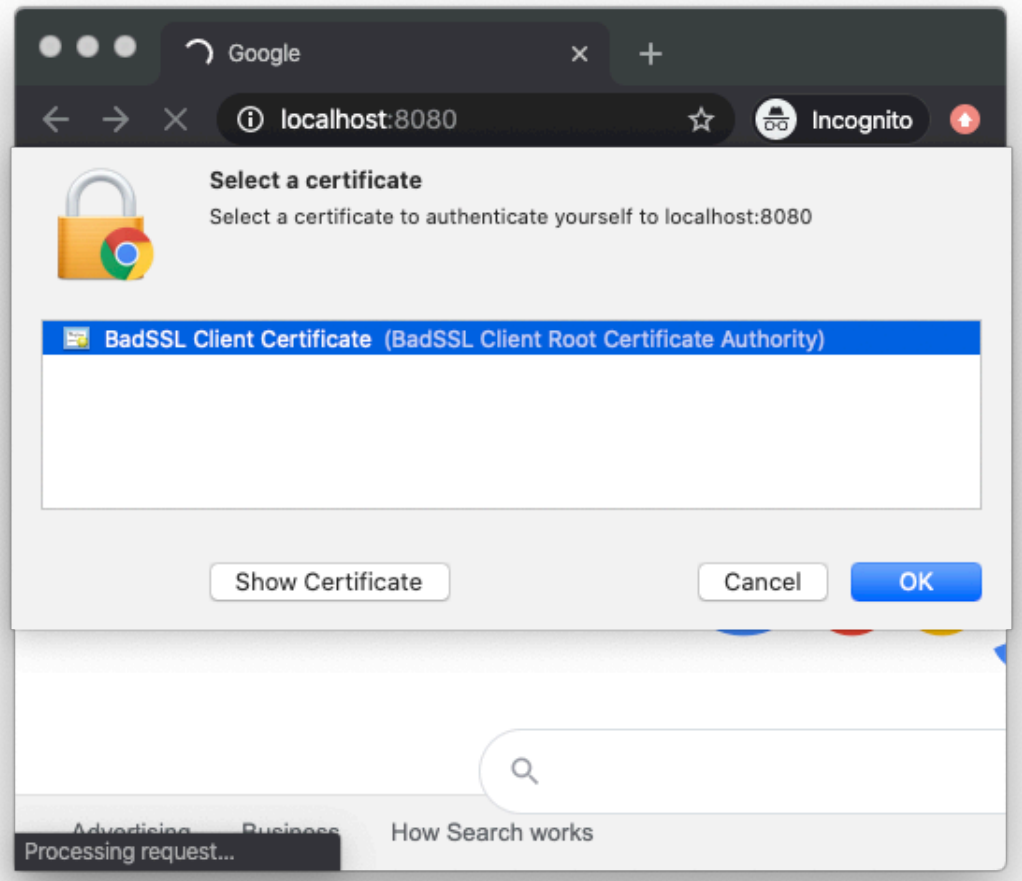


Figure 11.13 A page prompting the user's browser for a client certificate. The user can then select which certificate to use from a list of locally installed certificates. In the TLS handshake, the client certificate's key is then used to sign the handshake transcript, including the client's ephemeral public key used as part of the handshake.

Client-side certificates are pretty straightforward to understand. For example, in TLS 1.3, a server can request the client to authenticate during the handshake by sending a `CertificateRequest` message. The client then responds by sending its certificate (in a `Certificate` message) followed by a signature of all messages sent and received (in a `CertificateVerify` message). The signature of course includes the ephemeral public key used in the handshake's Diffie-Hellman key exchange. The client is authenticated if the server can recognize the certificate and successfully verify the client's signature.

Another example is the Secure Shell (SSH) protocol, which also have the client sign parts of the handshake with a public key known to the server.

Note that signing is not the only way to authenticate with public-key cryptography during the handshake phase. The Noise protocol framework (covered in chapter 9 as well) has several

handshake patterns that enable client-side authentication using just Diffie-Hellman key exchanges.[23]

POST-HANDSHAKE USER AUTHENTICATION WITH FIDO2

The second type of authentication with asymmetric keys use an already secure connection where only the server is authenticated.

To do this, a server can simply ask the client to sign a random challenge (this way replay attacks are prevented).

One interesting standard in this space is the **Fast IDentity Online 2 (FIDO2)**. FIDO2 is an open standard that defines how to use asymmetric keys to authenticate users. The standard specifically targets phishing attacks, and for this reason FIDO2 is made to work only with **hardware authenticators**. Hardware authenticators are simply physical components that can generate and store signing keys, and sign arbitrary challenges.

FIDO2 is split into two specifications:

- **Client-to-Authenticator Protocol (CTAP)**. CTAP specifies a protocol that **roaming authenticators** and **clients** can use to communicate with one another. Roaming authenticators are hardware authenticators that are external to your main device (think security keys, like the one pictured in figure 11.14). A client in the CTAP specification is defined as the software that wants to query these authenticators as part of an authentication protocol. Thus a client can be an operating systems, a native application like a browser, and so on.
- **Web Authentication (WebAuthn)**. WebAuthn is the protocol that web browsers and web applications can use to authenticate users with hardware authenticators. It thus must be implemented by browsers to support authenticators. If you are building a web application and want to support user authentication via hardware authenticators, WebAuthn is what you need to use. The standard allows websites to use not only roaming authenticators, but also "platform" authenticators. Platform authenticators are built-in authenticators provided by a device. They are usually implemented differently by different platforms, and often protected by biometrics (for example, a fingerprint reader, facial recognition, and so on).

Figure 11.14 A YubiKey is a security key, one type of "roaming" authenticator that can be used with FIDO2. Its role is to store private keys associated with different applications. It is inserted in the USB port of a computer (roaming authenticators can communicate with a platform via USB, Near Field Communication (NFC), or Bluetooth) and will not perform any cryptographic operations unless triggered by the touch of a finger on its gold metal button.

From a cryptography perspective, what is interesting to us is how applications can use these authenticators, or the asymmetric keys carried by them, to authenticate users. At a high level, applications can register a user's authenticator by asking it to create a new unique-to-the-application private key, and advertise the associated public key back to the application.

Applications are free to design their own authentication protocol on top of these authenticators, and implement FIDO2's CTAP to communicate with them. Let's see how WebAuthn deals with web app authentication:

1. Alice wishes to register a security key as a way to authenticate to `example.com` (this is called a "registration ceremony"). She clicks on some button on the website that lets her do just that.
2. `example.com` generates a challenge (a random number) along with an application id (maybe `example.com` has different applications).
3. Alice's browser sends a registration request to the security key using CTAP. The request contains binding information (the origin `example.com`), and the given challenge.
4. The security key generates a new keypair for this origin and application id, then replies with a signature over both the challenge and the origin, and the public key newly created.
5. The browser relays the public key and signature to `example.com` who can verify the signature and store Alice's public key.

And the login flow:

1. Alice wishes to log into `example.com` with a security key that she previously registered with the website (this is called an "authentication ceremony").
2. `example.com` (called the "relaying party") generates a challenge and send it to the browser.
3. Alice's browser sends an authentication request to the security key using CTAP, the request contains binding information (the origin `example.com`), and the given challenge.
4. The security key replies with a signature over both the challenge and the origin.
5. The browser relays the signature to `example.com`, who can verify the signature with Alice's public key (that it had stored at registration).

I illustrate this flow in figure 11.15.

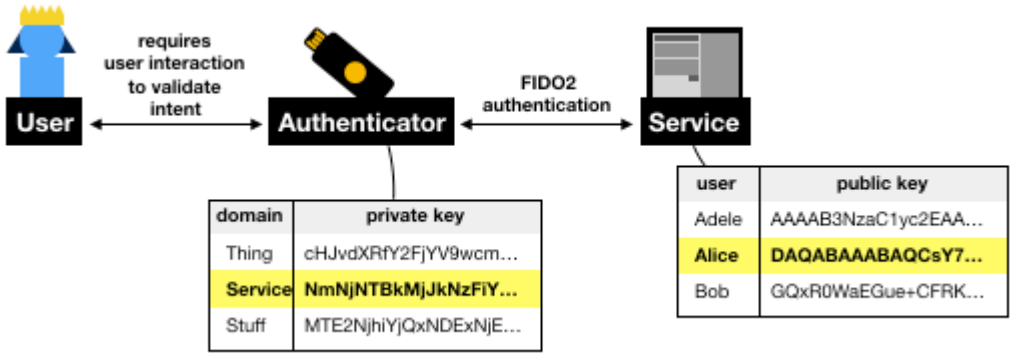


Figure 11.15 FIDO2 standardizes protocols for services to talk to authenticators, in order to provide user authentication only if the user consented to it it.

We are now ending the first part of this chapter. I recapitulate the non-password-based authentication protocols I've talked about in figure 11.16.

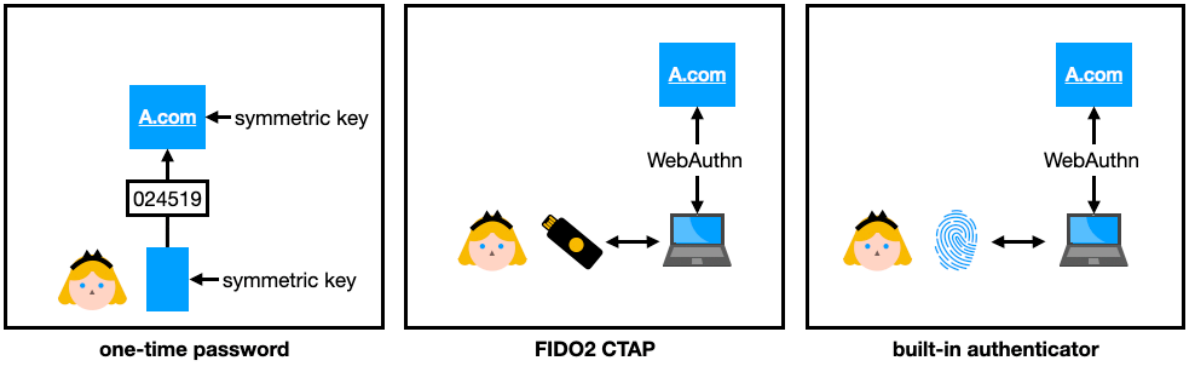


Figure 11.16 To authenticate without using a password, applications can allow users to either use symmetric keys via one-time password (OTP) based protocols, or use asymmetric keys via roaming authenticators or built-in authenticators.

Now that you have learned about many different techniques and protocols that exist to either improve passwords, or replace them with stronger cryptographic solutions, you might be

wondering which one you should use. Especially, each of these solutions have their own caveats, and no single solution might do it. If so, combine multiple ones! This idea is called **multi-factor authentication (MFA)**. Actually, chances are that you might have already used OTPs or FIDO2 as a second authentication factor, in addition to (and not in place of) passwords.

Next, let's take a look at how humans can help devices to authenticate each other.

11.3 User-aided authentication, pairing devices using some human help

Humans help machines to authenticate one another every day. EVERY DAY!

You've done it by **pairing** your wireless headphones with your phone, or by pairing your phone with your car, or by connecting some device to your home WiFi, and so on. And as with any pairing stuff, what's underneath is most probably a key exchange.

The authentication protocols in the last section took place in already-secured channels (perhaps with TLS) where the server was authenticated.

Most of this section, in contrast, attempts to provide a secure channel to two devices that do not know how to authenticate each other. In that sense, what you'll be learning in this section is how humans can help to **upgrade an insecure connection into a mutually-authenticated connection**. For this reason, the techniques you will learn about next should be reminiscent of some of the trust establishment techniques in the end-to-end protocols of chapter 10 (except that there, two humans were trying to authenticate themselves to each other).

Nowadays the most common insecure connections that you will run into, that do not go through the internet, are protocols based on short-range radio frequencies like Bluetooth, WiFi, and Near Field Communication (NFC). Devices that use these communication protocols tend to range from low-power electronics to full-featured computers.

This already sets some constraints for us:

- The device you are trying to connect to might not offer a screen to display a key, or a way to manually enter a key (we call this **provisioning** the device). For example, most wireless audio headsets today only have a few buttons and that's it.
- As a human is part of the validation process, having to type or compare long strings is often deemed impractical and not very user-friendly. For this reason many protocols attempt to shorten security-related strings to 4 or 6 digits PINs. (Imagine a protocol where you have to enter the correct 4-digit PIN to securely connect to a device, what are the chances to pick a correct PIN by just guessing?)

You're probably thinking back at some of your device pairing experiences, and realizing now that a lot of them **just worked™**:

1. You pushed some button on some device.
2. It entered pairing mode.
3. You then tried to find the device in the bluetooth device list on your phone.
4. After clicking on it, it successfully paired the device with your phone.

If you've read chapter 10, this should remind you of **trust on first use (TOFU)**, except that this time we also have a few more cards in our hand:

- **Proximity**. Both devices have to be close to each other. If using the NFC protocol (see figure 11.17 for a use case example), the devices have to be really close to each other.
- **Time**. Device pairing is often time-constrained. It is common that if, for example, in a 30 second window the pairing is not successful, the process must be manually restarted.

Figure 11.17 Near Field Communication (NFC) is a set of wireless protocols that allow devices to communicate when brought within 10cm of each other. The protocol is most commonly used by "contactless" payment systems (mobile payment applications like in the picture, contactless credit cards, and so on).

Unlike TOFU though, these real life scenarios usually do not allow you to validate after the fact that you've connected to the right device. This is of course not ideal, and one should strive for better security if possible.

> **NOTE** By the way this is how the Bluetooth core specification actually calls the TOFU-like protocol: "Just Works". I should mention that all built-in Bluetooth security protocols are currently broken, due to many attacks, including the latest KNOB attack released in 2019 (https://knobattack.com) The techniques surveyed in this chapter are nonetheless secure if designed and implemented correctly.

What's next in our toolkit? This is what we will see in this section: ways for a human to help devices to authenticate themselves.

Spoiler alert:

1. You'll see that cryptographic keys are always the most secure approach, but not necessarily the most user-friendly one.
2. You'll learn about symmetric PAKEs and how you can input the same password on two devices to connect them securely.
3. Finally you'll learn about protocols based on short authenticated strings (SAS) which authenticate a key exchange after the fact, by having you compare and match two short strings displayed by the two devices.

Let's get started!

11.3.1 Pre-shared keys

Naively, the first approach to connect a user to a device would be to reuse protocols that you've learned about in chapter 9 or chapter 10 (for example TLS or Noise), and to provision both devices with a symmetric shared secret, or better with long term public keys (in order to provide forward secrecy to future sessions).

This means that you need two things for each device to learn the other device's public key:

- You need a way to **export** a public key from its device.
- You need a way for a device to **import** public keys.

As we will see, this is not always straightforward or user-friendly.

But remember, we have a human in the mix, who can observe and maybe play with the two devices. This is unlike other scenarios that we've seen before. We can use this to our advantage! All the protocols that are to follow will be based on the fact that you have an additional **out-of-band** channel (you, the human in charge) that allows you to securely communicate some information.

The Authentication Problem - One of the main issues in cryptography is the establishment of a secure peer-to-peer (or group) communication over an insecure channel. With no assumption, such as availability of an extra secure channel, this task is impossible. However, given some assumption(s), there exists many ways to set up a secure communication.

– Sylvain Pasini Secure Communication Using Authenticated Channels (2009)

The addition of this out-of-band channel can be modeled as the two devices having access to two types of channels:

- An insecure channel. Think about a bluetooth or a WiFi connection with a device. By default, the user has no way of authenticating the device and can thus be man-in-the-middled.
- An authenticated channel. Think about a screen on a device. The channel provides integrity/authenticity of the information communicated, but poor confidentiality (someone could be looking over your shoulder).

I illustrate this in figure 11.18.

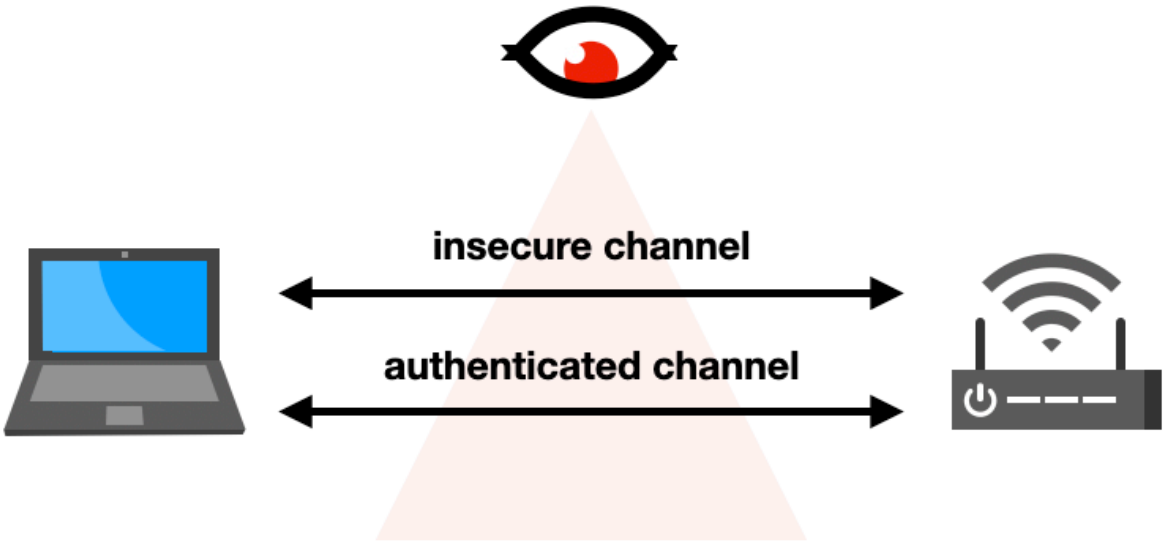


Figure 11.18 User-aided authentication protocols, that allow a human user to pair two devices, are modeled with two types of channels between the devices: an insecure channel (for example, NFC, Bluetooth, WiFi, and so on) that we assume is adversary controlled and an authenticated channel (for example, real life) that does not provide confidentiality but can be used to exchange relatively small amounts of information.

As this out-of-band channel provides poor confidentiality, we usually do not want to use it to export secrets, but rather public data. For example, a public key or some digest can be displayed by the device's screen.

Once you have exported a public key, you still need the other device to import it. For example, if the key was a QR code then the other device might be able to scan it, or if the key was encoded in a human-readable format then the user could manually type it in the other device using a

keyboard.

Once both devices are provisioned with each other's public keys, you can use any protocols I've mentioned in chapter 9 to perform a mutually-authenticated key exchange with the two devices.

What I want you to get from this section is that using cryptographic keys in your protocol is always the most secure way to achieve something, but that it is not always the most user-friendly way. Yet, real-world cryptography is full of compromise and trade-offs, and this is why the next two schemes not only exist but are the most popular ways to authenticate devices.

So let's see how **passwords** can be used to bootstrap a mutually-authenticated key exchange in cases where you cannot export and import long public keys, and then let's see how **short authenticated strings** can help when importing data is just not possible.

11.3.2 Symmetric password-authenticated key exchanges with CPace

The previous solution is what you should be doing if possible, as it relies on strong asymmetric keys as a root of trust. Yet, it turns out that in practice, typing a long string representing a key with some cumbersome keypad manually is not very user-friendly. What about these dear passwords? They are so much shorter, and thus easier to deal with. We love passwords right? Perhaps we don't, but users do, and real-world cryptography is full of compromises. So be it.

In the section on asymmetric password-authenticated key exchanges, I mentioned that a symmetric (or balanced) version exists where two peers who know a common password can perform a mutually-authenticated key exchange. This is exactly what we need.

Composable Password Authenticated Connection Establishment (CPace) was proposed in 2008 by Björn Haase and Benoît Labrique, and was chosen in early 2020 as the official recommendation of the CFRG.

The algorithm is currently being standardized as an RFC. The protocol, simplified, looks like this:

1. The two devices derive a generator (for some predetermined cyclic group) based on the common password.
2. The two devices use this generator to perform an ephemeral Diffie-Hellman key exchange on top of it.

I illustrate the algorithm in figure 11.19.

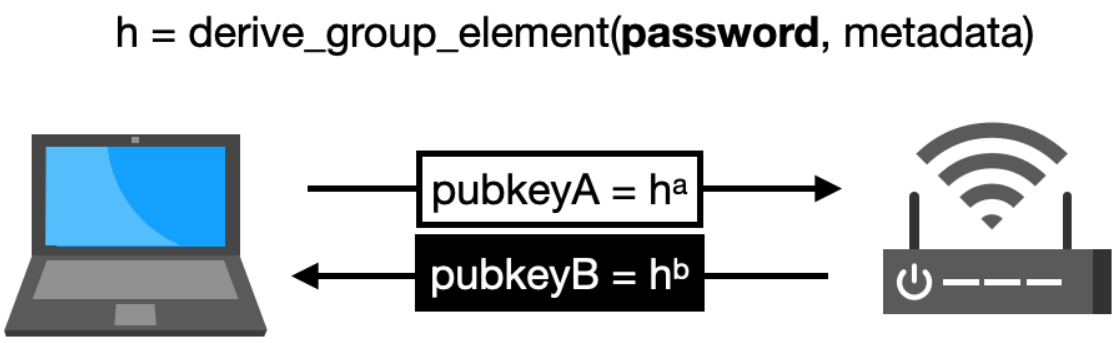


Figure 11.19 The CPace PAKE works by having the two devices generate a generator based on the password, and use it to perform as base for the usual ephemeral Diffie-Hellman key exchange.

The devil is in the details of course, and as a modern specification CPace targets elliptic curve gotchas and defines when one must verify that a received point is in the right group (due to the trendy Curve25519 that unfortunately does not span a prime group). It also specifies how one derives a generator based on a password when in an elliptic curve group (using so-called hash-to-curve algorithms), and how to do it using not only the common password but a unique session id and some additional contextual metadata like IP addresses of the peers and so on. These steps are important as both peers must derive a generator h in a way that prevents them from knowing its discrete logarithm x such that $g^x = h$. Finally the session key is derived from the Diffie-Hellman key exchange output, the transcript (the ephemeral public keys) and the unique session id.

Intuitively, you can see that impersonating one of the peers, and sending a group element as part of the handshake, means that you're sending a public key which is associated to a private key you cannot know. This means essentially that you can never perform the Diffie-Hellman key exchange if you don't know the password. The transcript just looks like a normal Diffie-Hellman key exchange and so no luck there as well as long as Diffie-Hellman is secure.

11.3.3 Was my key exchange man-in-the-middled? Just check a short authenticated string (SAS)

In the second part of this chapter, you've seen different protocols that allow two devices to be paired with the help of a human. Yet, I've mentioned that some devices are so constrained that they cannot make use of them. Let's take a look at a scheme that is used when the two devices cannot import keys, but can display some limited amount of data to the user. Perhaps via a screen, or by turning on some LEDs, or by emitting some sounds, and so on.

First, remember that in chapter 10 you learned about authenticating a session post-handshake (after the key exchange) using fingerprints (hashes of the transcript). We could use something like this, as we have our out-of-band channel to communicate these fingerprints. If the user can successfully compare and match the fingerprints obtained from both devices, then the user knows

that the key exchange was not man-in-the-middled. The problem with fingerprints is that they are long bytestrings (typically 32-byte long) which might be hard to display to the user, and are also cumbersome to compare. But for device pairing, we can use much shorter bytestrings as we are doing the comparison in real time!

We call these **short authenticated strings (SAS)**. SAS are used a lot, notably by Bluetooth, due to them being quite user-friendly (see figure 11.20 for an example).

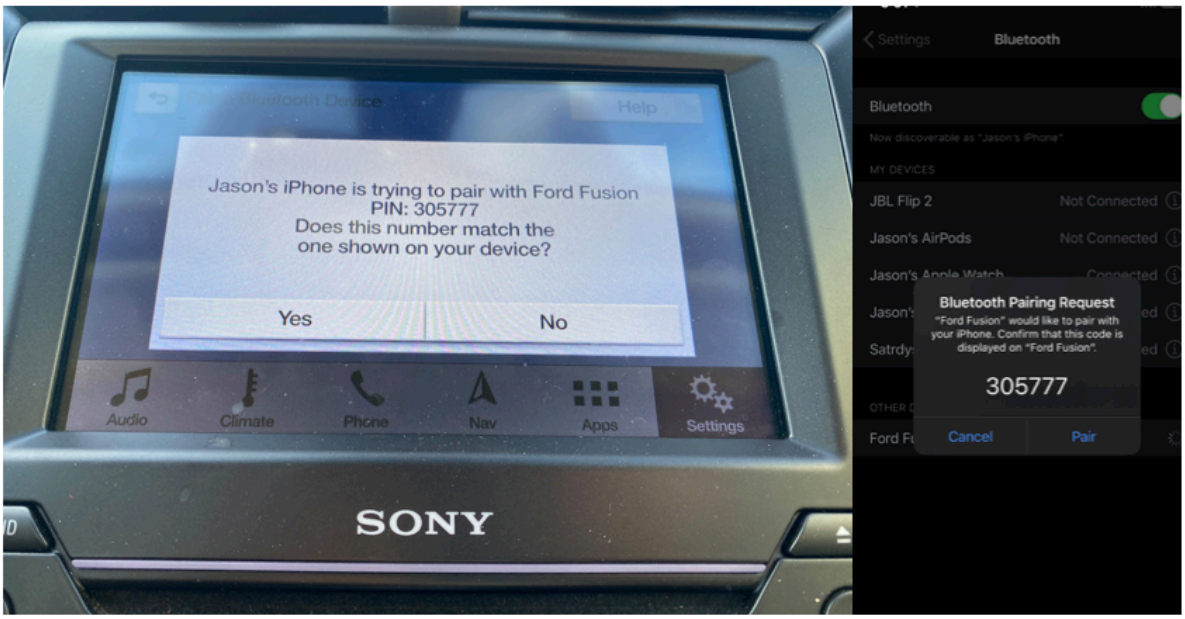


Figure 11.20 To pair a phone with a car via Bluetooth, the "numeric comparison" mode can be used to generate a short authenticated string (SAS) of the secure connection negotiated between the two devices. Unfortunately, as I stated earlier in this chapter, due to the KNOB attack Bluetooth's security protocols are currently broken (as of 2020). If you control both devices, you need to implement your own SAS protocol.

There aren't any standards for SAS-based schemes, but most protocols (including Bluetooth's numeric comparison) implement a variant of the **Manually Authenticated Diffie-Hellman (MA-DH)**.[24] MA-DH is a simple key exchange with an additional trick that makes it hard for an attacker to actively man-in-the-middle the protocol.

But you might ask, why not just create a SAS from truncating a fingerprint? Why the need for a trick?

A SAS is typically a 6-digit number, which can be obtained by truncating a hash of the transcript to less than 20 bits and converting that to numbers in base 10. A SAS is thus dangerously small, which makes it much easier for an attacker to obtain a **second pre-image** on the truncated hash. In figure 11.21, we take the example of two devices (although we use Alice and Bob) performing an unauthenticated key exchange. An active man-in-the-middle attacker can substitute Alice's public key with their own public key in the first message. Once the attacker receives Bob's

public key, they know what SAS Bob will compute (a truncated hash based on the attacker's public key and on Bob's public key). So the attacker just has to generate many $public\_key_{E2}$ (before sending it to Alice) in order to find one that will make the SAS of Alice match with Bob's.

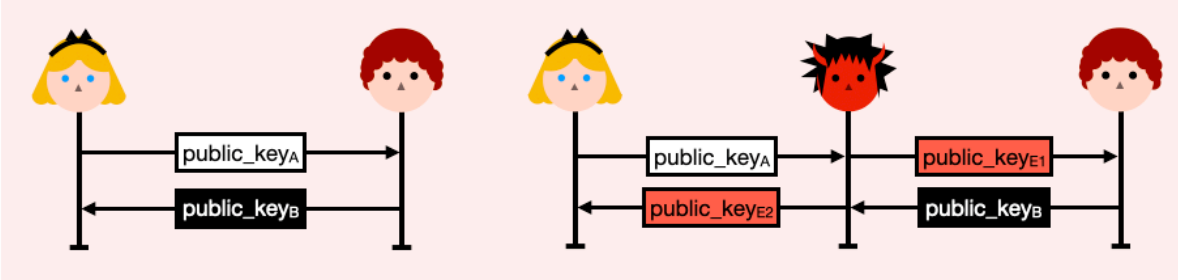


Figure 11.21 A typical unauthenticated key exchange (diagram on the left) can be intercepted by an active man-in-the-middle attacker (diagram on the right) who can then substitute the public keys of both Alice and Bob. A man-in-the-middle attack is successful if both Alice and Bob generate the same short authenticated string. That is, if hash($public\_key_A$ || $public\_key_{E2}$) and hash($public\_key_{E2}$ || $public\_key_B$) match.

Generating a public key to make both SAS match is actually pretty easy. Imagine that the SAS is 20 bits, then after only 2^{20} computations you should find a second pre-image that will have both Alice and Bob generate the same SAS. This should be pretty instant to compute, even on a cheap phone.

The trick behind SAS-based key exchanges is to prevent the attacker from being able to choose their second public key to force the two SAS to match. To do this, Alice simply sends a **commitment** of her public key before seeing Bob's public key (as in figure 11.22).

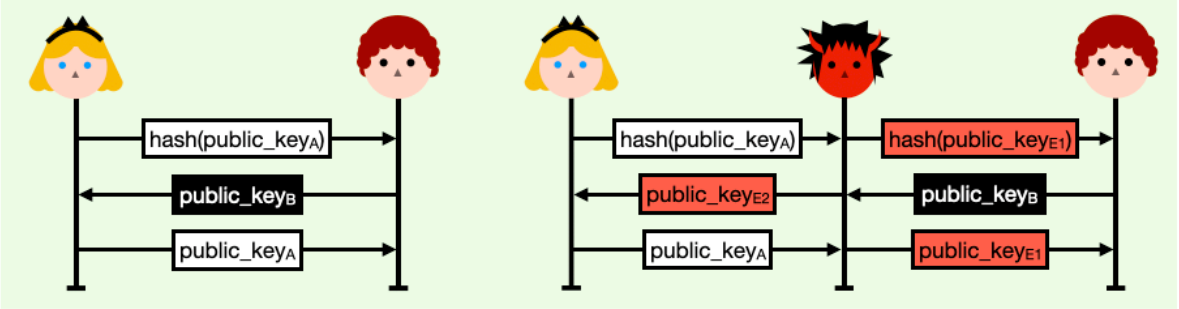


Figure 11.22 The diagram on the left pictures a secure SAS-based protocol in which Alice first sends a commitment of her public key. She then only reveals her public key after receiving Bob's public key. As she committed to it, she cannot freely choose her keypair based on Bob's key. If the exchange is actively man-in-the-middled (diagram on the right), the attacker cannot choose either keypairs to force Alice and Bob SAS to match.

As with the previous insecure scheme, the attacker's choice of $public\_key_{E1}$ does not give them any advantage. But now, they also cannot choose a $public\_key_{E2}$ that helps, as they do not know

Bob's SAS at this point in the protocol. They are thus forced to shoot in the dark, and hope that Alice's and Bob's SAS will match. If a SAS is 20 bits, that's a probability of 1 out of 1,048,576. An attacker can have more chances by running the protocol multiple times, but keep in mind that every instance of the protocol must have the user manually match a SAS. Effectively, this friction naturally prevents an attacker from having too many lottery tickets.

> **NOTE** Interestingly, as I was writing chapter 10 on end-to-end encryption, I started looking into how users of the Matrix end-to-end encrypted chat protocol authenticated their communications. In order to make the verification more user-friendly, Matrix created their own variant of a SAS-based protocol. Unfortunately, they hashed the shared secret of an X25519 key exchange, and did not include the public keys being exchanged in the hash. In chapter 5, I mentioned that it is important to validate X25519 public keys. Matrix did not, and this allowed a man-in-the-middle attacker to send incorrect public keys to both users and force them to end up with the same predictable shared secret, and in turn the same SAS. This completely broke the end-to-end encryption claim of the protocol, and ended up being quickly fixed by Matrix.

This is it! Figure 11.23 recapitulate the different techniques you learned in the second part of this chapter. I'll see you in chapter 12.

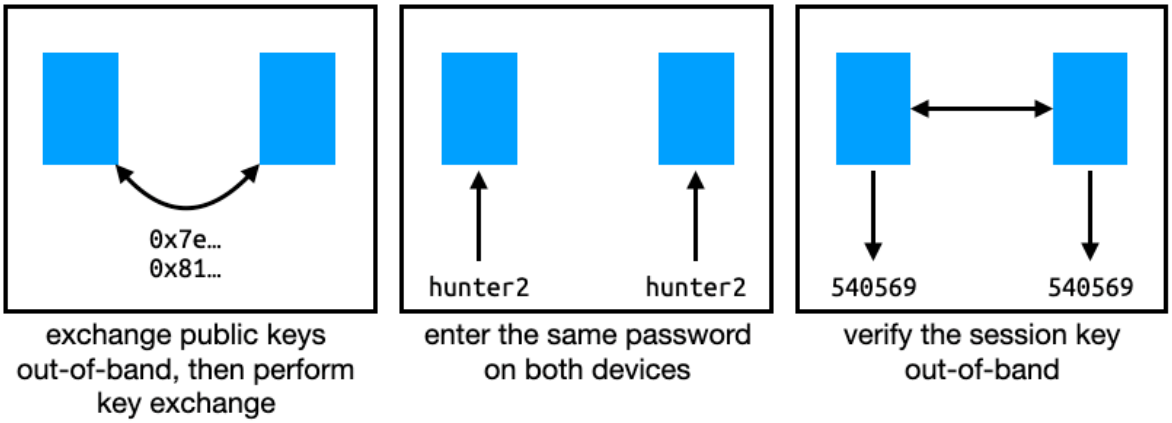


Figure 11.23 You've learned about three techniques to pair two devices: 1. a user can either help the devices obtain each other's public keys so that they can perform a key exchange 2. a user can enter the same password on two devices so that they can perform a symmetric password-authenticated key exchange 3. a user can verify a fingerprint of the key exchange after the fact, to confirm that no man-in-the-middle attacker intercepted the pairing.

11.4 Summary

- User authentication protocols (Protocols for machines to authenticate humans) often take place over secure connections, where only the machine (server) has been authenticated. In this sense, it upgrades a one-way authenticated connection to a mutually-authenticated connection.
- User authentication protocols make heavy use of passwords. Passwords have proven to be a somewhat practical solution that has been widely accepted by users, but has also led to many issues due to poor password hygiene, low entropy, and password database breaches.
- There are two ways to avoid having users carry multiple passwords (and possibly reuse passwords):
    - Password Managers, tools that users can use to generate (and manage) strong passwords for every application they use.
    - Single-sign on (SSO), a federated protocol that allows a user to use one account to register and log into other services.
- A solution for servers to avoid learning about and storing their users' passwords, is to use an asymmetric password-authenticated key exchange (asymmetric PAKE). An asymmetric PAKE like OPAQUE allows users to authenticate to a known server using passwords, but without having to actually reveal their passwords to the server.
- Solutions to avoid passwords altogether are for users to use symmetric keys via one-time passwords (OTP) algorithms or use asymmetric keys via standards like FIDO2.
- User-aided authentication protocols often take place over insecure connections (WiFi, Bluetooth, NFC) and help two devices to authenticate each other. To secure connections in these scenarios, user-aided protocols assume that the two participants possess an additional authenticated (but not confidential) channel that they can use (for example a screen on the device).
- Exporting a device public key to another device could allow strongly mutually-authenticated key exchanges to happen. These flows are unfortunately not very user-friendly, and sometimes not possible due to device constraints (no way to export or import keys).
- Symmetric password-authenticated key exchanges (symmetric PAKEs) like CPace can decrease the burden for the user to import a long public key, by only having to manually input a password in a device. Symmetric PAKEs are already used by most people to connect to their home WiFi for example.
- Protocols based on short authenticated strings (SAS) can provide security for devices that cannot import keys or passwords, but are able to display a short string after a key exchange has taken place. This short string has to be the same on both devices in order to ensure that the unauthenticated key exchange was not actively man-in-the-middled.

12 Crypto as in cryptocurrency?

This chapter covers

- What cryptocurrencies are and how they were made possible thanks to consensus algorithms.
- The different types of cryptocurrencies that exist.
- How the Bitcoin and Diem cryptocurrencies work in practice.

For as far as I can remember, the term "crypto" has been used in reference to the field of cryptography. Recently, I have seen its meaning quickly changing and being used by more and more people to refer to **cryptocurrencies**. Cryptocurrency enthusiasts, in turn, seem to get more and more interested to learn about cryptography. Which makes sense, as cryptography is at the core of cryptocurrencies.

But first, what's a cryptocurrency? It is two things:

- It's a **digital currency**. Simply: it allows people to transact some currency electronically. Sometimes a currency backed by a government is used (like the US dollar), and sometimes a made-up currency is used (like bitcoin). You likely already use digital currencies—whenever you send money to someone on the internet or use a checking account, you are using a digital currency! Indeed, you don't need to send cash by mail anymore and most money transactions today are just updates of rows in databases.
- It's a currency that relies heavily on cryptography to avoid having to use a trusted third party and to provide transparency. In a cryptocurrency there is no central authority that one has to blindly trust (like a government or a bank). We often talk about this property as **decentralization**, as in "we are decentralizing trust." Thus, as you will see in this chapter, cryptocurrencies are designed to tolerate a certain number of malicious actors, and to allow people to verify their well-functioning.

Cryptocurrencies are relatively new, as the first experiment to be successful was Bitcoin which

was proposed in 2008. It was during the same year as the 2008 global financial crisis, which was started by US banking institutions and then spread to the world, eroding the trust people had in the financial system and giving a platform to more transparent initiatives like Bitcoin. At that time, many people started to realize that the status quo for financial transactions was inefficient, expensive to maintain, and opaque to the people. The rest is history, and I believe this book is the first book on cryptography to include a chapter on cryptocurrencies.

12.1 A gentle introduction to Byzantine fault-tolerant consensus algorithms

Imagine that you want to create a new digital currency. It's actually not too involved to build something that works. You could set up a database on a dedicated server, which would be used to track users and their balances. With this, you could provide an interface for people to query their balance or let them send payments (which would reduce their balance in the database and increase the balance of another row). Initially, you could also randomly attribute some of your made-up currency to your friends, so that they can start transfering money on your system.

But such a simple system has a number of flaws.

12.1.1 A problem of resilience - distributed protocols to the rescue

The system we just saw is a **single point of failure**. If you lose electricity, you users won't be able to use the system. Worse, if some natural disaster unexpectedly destroys your server, everybody might permanently lose their balance. To tackle this issue, there exist well-known techniques that can be used to provide more resilience to your system. The field of "distributed systems" studies such techniques. In this case, the usual solution used by most large applications is to replicate the content of your database in (somewhat) real time to other backup servers. These servers can then be distributed across various geographical locations, ready to be used as backup or even to take over if your main server goes down (this is called **high availability**). You now have a "distributed database".

For very large systems that serve many many queries, it is often the case that these backup databases are not just sitting on the sideline waiting to be useful, but instead used to provide reads to the state. It is difficult to have more than one database accept writes and updates, because then you could have conflicts (the same way two people editing the same document can be dangerous). Thus, you often want a single database to act as "**leader**" and order all writes and updates to the database, while others can still be used to read the state.

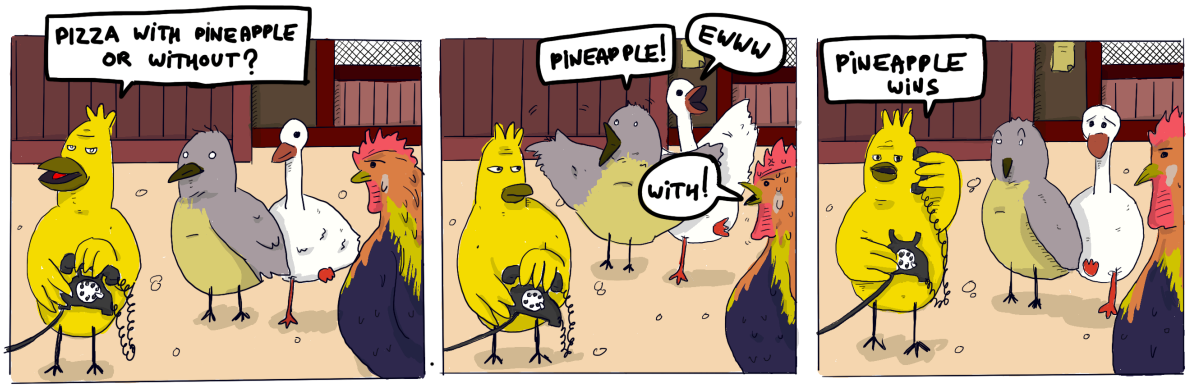
Replication of database content can be slow, and it is expected that some of your databases will lag behind the leader until they can catch up. This is especially true if they are situated further

away in the world, or are experiencing network delays due to other reasons. This lag becomes a problem when the replicated databases are used to read the state (imagine that you can see a different account balance than your friend, because you are both querying different servers).

In these cases, applications are often written in order to tolerate this lag. This is referred to as **eventual consistency**, as eventually the states of the databases become consistent. (Stronger consistency models exist, but they are usually very slow and impractical.)

Such systems have other problems: if the main database crashes, who gets to become the main database? Another problem is: if the backup databases were lagging behind when the main database crashed, do we lose some of the latest changes?

This is where stronger algorithms, **consensus algorithms**,[25] come into place: when you need the whole system to agree (or come to consensus) on some decision. Imagine that a consensus algorithm solves the solution of a group of people trying to agree on what pizza to order. It's easy to see what the majority wants if everyone is in the same room, but if everyone is communicating through the network where messages can be delayed, dropped, intercepted, and modified, then a more complicated protocol is required.



Let's see how consensus can be used to answer the previous two questions. The first question of who gets to take over in the case of a crash is called "leader election", and a consensus algorithm is often used to get all the databases to agree on who will be the next leader. The second question is often solved by seeing database changes in two different steps: pending and committed. Changes to the database state are always pending at first, and can only be set as committed if "enough" of the databases agreed to commit it (this is where a consensus protocol can be used as well). Once committed, the update to the state cannot be lost easily as most of the participants have committed the change.

Some well-known consensus algorithms are Paxos (published by Lamport in 1989) and its following simplification Raft (published by Ongaro and Ousterhout in 2013).[26] They are used in most distributed database systems to solve different problems.

12.1.2 A problem of trust - decentralization helps

Distributed systems provide a resilient alternative to systems that act as a single point of failure, that is from an operational perspective. The consensus algorithms used by most distributed database systems do not tolerate faults well. As soon as machines start crashing, or start misbehaving (hardware faults), or start getting disconnected from some of the other machines (network partitions), problems arise. There's also no ways to detect this from a user perspective, which is even more of an issue if servers become compromised.

If I query a server and it tells me that Alice has 5 billion dollars in her account, I'll just have to trust it. Perhaps the server could include in its response all the money transfers that she has received and sent since the beginning of time, and summing it all up I could verify that indeed it results with the 5 billion dollars she has in her account. But what tells me the server didn't lie to me? Maybe when Bob asks a different server, it'll return a completely different balance and/or history for Alice's account. We call this a **fork**: two contradicting states that are being presented as valid. A branch in history that should never have happened. And thus, you can imagine that the compromise of one of the replicated databases can lead to pretty devastating consequences.

In chapter 9, I mentioned Certificate Transparency, a gossip protocol that aims at detecting such forks in the web public key infrastructure. The problem with money is that detection only is not enough. You want to prevent the problem from happening in the first place.

In 1982, Lamport (the author of the Paxos consensus algorithm) introduced the idea of **Byzantine fault-tolerant (BFT) consensus algorithms**.

> *We imagine that several divisions of the Byzantine army are camped outside an enemy city, each division commanded by its own general. The generals can communicate with one another only by messenger. After observing the enemy, they must decide upon a common plan of action. However, some of the generals may be traitors, trying to prevent the loyal generals from reaching agreement.*
>
> *– Lamport et al. The Byzantine Generals Problem (1982)*

With his byzantine analogy, Lamport started the field of BFT consensus algorithms, aiming at preventing bad consensus participants from creating different conflicting views of a system when agreeing on a decision. These BFT consensus algorithms highly resemble previous consensus algorithms like Paxos and Raft, except that of course the replicated databases (the participants of the protocol) do not blindly trust one another anymore. BFT protocols usually make heavy use of cryptography to authenticate messages and decisions, which in turn can be used by others to cryptographically validate the decisions output by the consensus protocol.

These BFT consensus protocols are thus solutions to both our resilience and trust issues: the different replicated databases can run these BFT algorithms to agree on new states for the system (for example, user balances), while policing each other by verifying that both the state transitions

(transactions between users) are valid, and have been agreed on by most of the participants. We say that the trust is now **decentralized**.

The first practical BFT consensus algorithm invented was **Practical BFT (PBFT)**, published in 1999. PBFT is a leader-based algorithm, similar to Paxos and Raft, where one leader is in charge of making proposals while the rest attempts to agree on these proposals. Unfortunately PBFT is quite complex, slow, and doesn't scale well (past a dozen participants). Today, most modern cryptocurrencies use more efficient variants of PBFT. For example Diem, the cryptocurrency introduced by Facebook in 2019, is based on HotStuff—a PBFT-inspired protocol.

12.1.3 A problem of scale - permissionless and censorship-resistant networks

One limitation of these PBFT-based consensus algorithms is that they all require a known and fixed set of participants. More problematic, past a certain number of participants they start breaking apart: communication complexity increases drastically, they become extremely slow, electing a leader becomes complicated, etc.

So how does a cryptocurrency decide who the consensus participants are? There are several ways, but the two most common ways are:

- **Proof of authority (PoA).** The consensus participants are decided in advance.
- **Proof of stake (PoS).** The consensus participants are picked dynamically based on who has the most at stake (and thus is less incentivized to attack the protocol). In general, cryptocurrencies based on PoS will elect participants based on the amount of digital currency they hold.

Having said that, not all consensus protocols are classical BFT consensus protocols. Bitcoin, for example, took a very different approach when it proposed a consensus mechanism that had no known list of participants. This was quite a novel idea at the time, and Bitcoin achieved this by relaxing the constraints of classical BFT consensus protocols. As you will see later in this chapter, because of that Bitcoin can fork, and it introduces its own sets of challenges.

Without participants, how do you even pick a leader? You could use a proof of stake system (for example, the Ouroboros consensus protocol does that), but instead Bitcoin's consensus relied on a probabilistic mechanism called **proof of work (PoW)**. In Bitcoin, this translates to people attempting to find solutions to puzzles in order to become a participant and a leader. Of course, the puzzle is a cryptographic one, as you will see later in this chapter.

Due to this lack of known list of participants, Bitcoin is called a **permissionless** network. In a permissionless network you do not need extra permissions to participate in consensus: anyone can participate. This is in contrast to **permissioned** networks that have a fixed set of participants. I summarize some of these new concepts in figure 12.1.

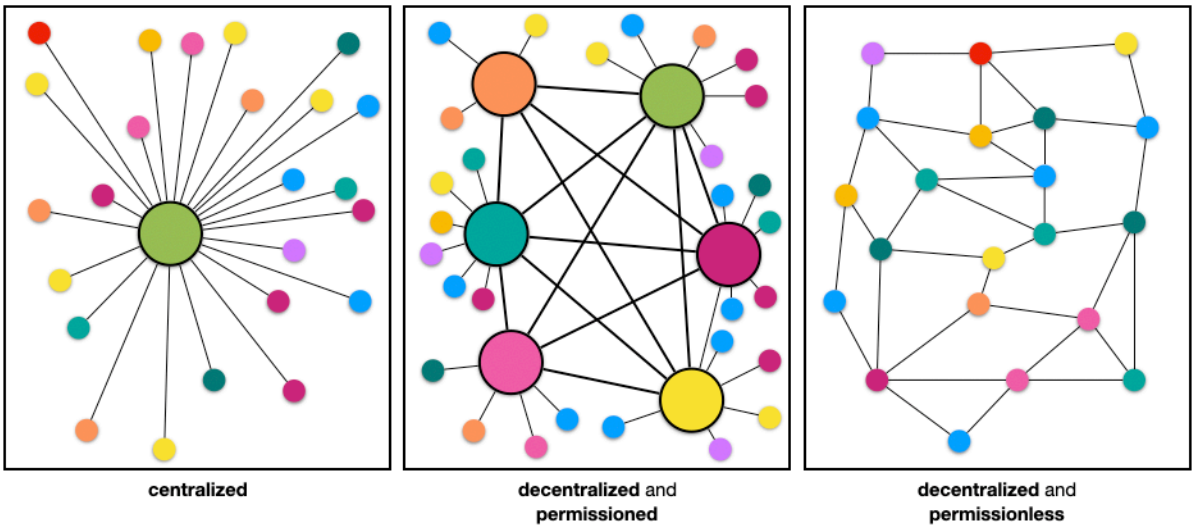


Figure 12.1 A centralized network can be seen as a single point of failure, whereas a distributed and decentralized network is resilient to a number of servers shutting down or acting malicious. A permissioned network has a known and fixed set of main actors, while in a permissionless network anyone can participate.

Up until recently, it was not known how to use classical BFT consensus protocols with a permissionless network where anyone is allowed to join, but today there exist many approaches using proof of stake to dynamically pick a smaller subset of the participants as consensus participants. One of the most notable one is Algorand, published in 2017, which dynamically picks participants and leaders based on how much currency they hold.

In addition, Bitcoin also claims to be resistant to censorship as you cannot know in advance who will become the next leader, and can't prevent the system from electing a new leader. It is less clear if this is possible in proof-of-stake systems where it might be easier to figure out the identities behind large sums of currency.

Finally, I should mention that not all BFT consensus protocols are leader-based. Some are **leaderless**, they do not work by having elected leaders decide on the next state transitions, instead everyone can propose changes and the consensus protocol helps everyone agree on the next state. In 2019, Avalanche launched such a cryptocurrency which allowed anyone to propose changes and participate in consensus.

In the rest of this chapter, I will go over two different cryptocurrencies in order to demonstrate different aspects of the field:

- **Bitcoin**. The most popular cryptocurrency based on proof-of-work, introduced in 2008.
- **Diem**. A cryptocurrency based on a BFT consensus protocol, and announced by Facebook and a group of other companies in 2019.

12.2 How does Bitcoin work?

On October 31st, 2008, an anonymous researcher(s) published "Bitcoin: A Peer-to-Peer Electronic Cash System" under the pseudonym Satoshi Nakamoto.[27] Not long after, they released the Bitcoin core client: a software that anyone can run in order to join and participate in the Bitcoin network. That's the only thing that Bitcoin needed: enough users that run the same software (or at least the same algorithm). The first ever cryptocurrency was born: the bitcoin (or BTC).

Bitcoin is a true success story. The cryptocurrency has been running for more than a decade (at the time of this writing), and has allowed users from all around the world to transact using the digital currency. In 2010, Laszlo Hanyecz, a developer, bought two pizzas for 10,000 BTC. 7 years later, the price of one BTC had reached $20,000. As I am writing these lines (February 2021), a bitcoin is worth almost $57,000. Thus, one can already take away that cryptocurrencies can sometimes be extremely volatile.

12.2.1 How Bitcoin handles user balances and transactions

Let's dive deeper into the internals of Bitcoin, first looking at how Bitcoin handles user balances and transactions.

As a user of Bitcoin, you directly deal with cryptography. You do not have a username and password to log into a website, like with any banks, instead you have an ECDSA keypair that you can generate yourself. A user's balance is simply an amount of bitcoin associated with a public key, and as such, to receive bitcoins you simply share your public key with others. To use your bitcoins, you sign a transaction with your private key. A transaction pretty much says what you think it says: "I send X BTC to public key Y", overlooking some details that I'll explain later.

> **NOTE** **In chapter 7, I mentioned that Bitcoin uses the secp256k1 curve with ECDSA. The curve is not to be confused with NIST's P-256 curve, which is known as secp256r1.**

Thus, the protection of one's holdings is directly linked to one's private key. And as you know, key management is hard, which has led to the accidental loss (or theft) of keys worth millions of dollars… Be careful!

There exist different types of transactions in Bitcoin, and most of the transactions seen on the network actually hide the recipient's public key by hashing it. In these cases, the hash of a public key is referred to as the "address" of an account. (For example, this is my Bitcoin address: `bc1q8y6p4x3rp32dz80etpyffh6764ray9842egchy`.) An address effectively hides the actual public key of the account, that is until the account owner decides to spend its bitcoins (in which

case they'll have to reveal the pre-image of their address so that others can verify the signature). This makes addresses shorter in size, and prevents someone from retrieving your private key in case ECDSA is broken one day.

The fact that different types of transactions exist is an interesting detail of Bitcoin: transactions are actually not just payloads containing some information, they are scripts written in a made-up and quite limited instruction set. When a transaction is processed, the script needs to be executed before the produced output can determine if the transaction is valid or not (and if it is, what steps need to be taken to modify the state of all the accounts). Cryptocurrencies like Ethereum have pushed this scripting idea to the limits by allowing much more complex programs to run when a transaction is executed (so-called "smart contracts").

There are two things here that I didn't touch on:

1. What's in a transaction?
2. What does it mean for a transaction to be executed? And who executes it?

I will explain the second item in the next section, so let's look at what really is in a transaction for now.

A particularity of Bitcoin is that there is no real database of account balances. Instead, a user has pockets of bitcoins that are available for them to spend, and which are called **Unspent Transaction Outputs (UTXOs)**. You can think of the concept of UTXOs as a large bowl, visible to everyone, and filled with coins that only their owners can spend. When a transaction spends some of the coins, the coins disappear from the bowl, and new ones appear for the payees of the same transaction. These new coins are just the outputs listed in the transaction.

To know how much bitcoins you have in your account, you'd have to count all of the UTXOs that are assigned to your address. In other words, you'd have to count all of the money that was sent to you, and that you haven't spent yet.

Figure 12.2 gives an example that illustrates how UTXOs are used in transactions.

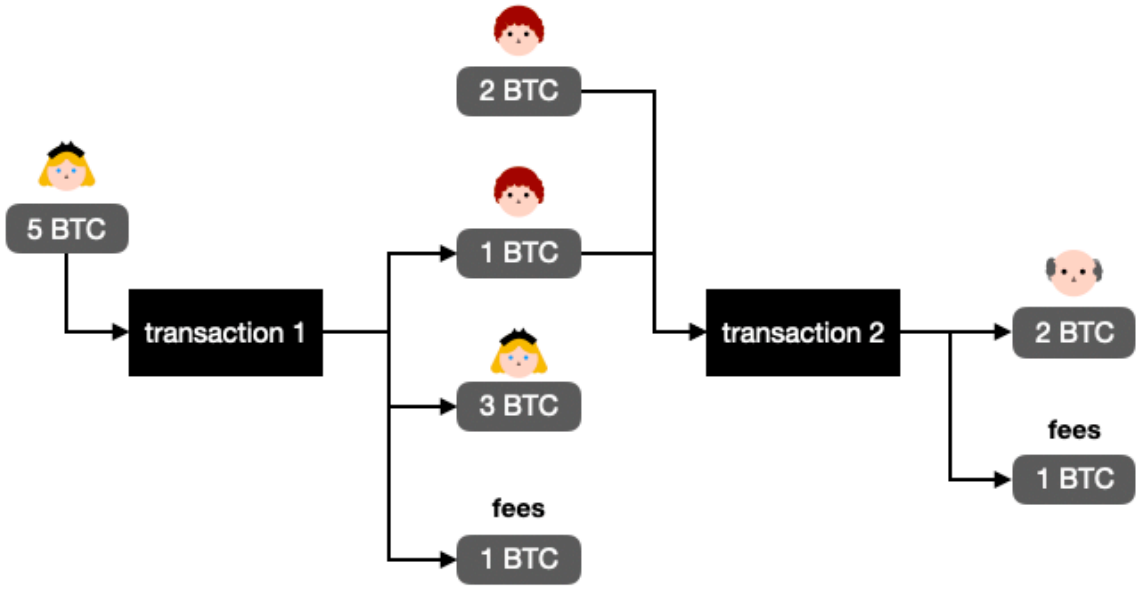


Figure 12.2 Transaction 1 is signed by Alice and transfers 1 BTC to Bob. Since it uses a UTXO of 5 BTC, it needs to also send back the change to Alice, as well as reserve some of that change as fees. Transaction 2 is signed by Bob and combines two UTXOs to transfer 2 BTC to Felix. Note that in reality fees are much much lower.

There's now a chicken and egg question to ask: where did the first UTXOs come from? That, I will answer in the next section.

12.2.2 Mining bitcoins in the digital age of gold

You now understand what's in a Bitcoin transaction, and how you can manage your account or figure out someone's balance. But who actually keeps track of all these transactions? The answer is everyone! Indeed, using Bitcoin means that every transaction must be publicly shared and recorded in history. Bitcoin is an **append-only** ledger—a book of transactions where each page is connected to the previous one. I want to emphasize here that append-only means that you can't go back and alter a page in the book. Note also that since every transaction is public, the only semblance of anonymity you get is that it might be hard to figure out who is who (in other words, what public key is linked to what person in real life).

One can easily inspect any transaction that has happened since the inception of Bitcoin by downloading a Bitcoin client and downloading the whole history. By doing that, you become part of the network and re-execute every transaction according to the rules encoded in the Bitcoin client. Of course, Bitcoin's history is pretty massive: at the time of this writing it is around 300 GB, and it can take days depending on your connection to download the entire Bitcoin ledger. You can more easily inspect transactions by using an online service that did the heavy lifting for you (as long as you trust an online service). I give an example of these so-called "blockchain explorers" in figure 12.3.

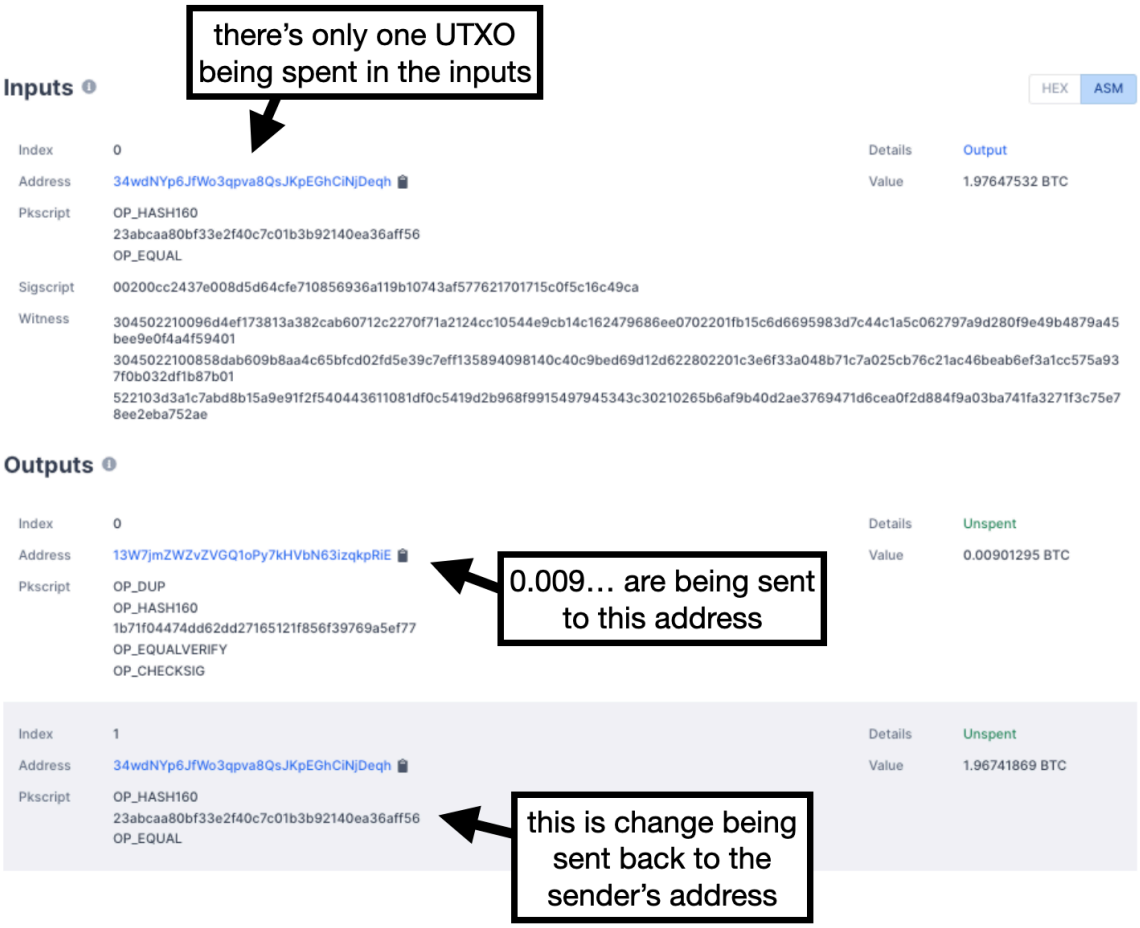


Figure 12.3 A random transaction I chose to analyze on https://blockchain.com/.../...f0b7b1270875bdbc4e1fd00ae755cb3b57c15adb0014c58d90. The transaction uses one input (of around 1.976 BTC) and splits it in two outputs (of around 0.009 BTC and 1.967 BTC). The difference between the total input amount and the total output amount is the transaction fee (not represented as an output). The other fields are the scripts written using Bitcoin's scripting language in order to either spend the UTXOs in the inputs, or to make the UTXOs in the outputs spendable.

Bitcoin is really just a list of all the transactions that have been processed since the very beginning (we call that the genesis) up until now. This should make you wonder: who is in charge of choosing and ordering transactions in this ledger?

In order to agree on an ordering of transactions, Bitcoin allows anyone (even you) to propose a list of transactions to be included in the next page of the ledger. This proposal containing a list of transactions is called a **block** in Bitcoin's terms. But letting anyone propose a block is a recipe for disaster, as there are a lot of participants in Bitcoin. Instead, we want just one person to make a proposal for the next block of transactions. To do this, Bitcoin makes everybody work on some probabilistic puzzle, and only allows the one who solves the puzzle first to propose their block. This is called **proof of work (PoW)**. Bitcoin's proof of work is based on finding a block that hashes to a digest smaller than some value. In other words, the blocks' digest must have a binary representation starting with some given numbers of zeros.

In addition to transactions you want to include, the block must contain the hash of the previous block. Hence the Bitcoin ledger is really a succession of blocks, where each block refers to the previous one, down to the very first block: the genesis block. This is what Bitcoin calls a **blockchain**. The beauty of the blockchain is that the slightest modification to a block would render the chain invalid, as the block's digest would also change and consequently break the reference the next block had to it.

Note that as a participant who is looking to propose the next block, you don't have to change much in your block to derive a new hash from it. You can fix most of its content first (the transactions it includes, the hash of the block it extends, etc.) and then only modify a field called the blocks' "nonce" to impact the block's hash. You can treat this field as a counter, just incrementing the value until you find a digest that fits the rules of the game, or generate a random value.

I illustrate this idea of a blockchain in figure 12.4.

Figure 12.4 On https://andersbrownworth.com/blockchain/blockchain one can interactively play with a toy blockchain. Each block includes its parent's digest, and each block contains a nonce found randomly that allows its digest to start with 4 zeros. Notice that this is true for the top blockchain, but the bottom one contains a block (number 2) that has been modified (its data was initially empty). As the modification changed the digest of the block, it is no longer being authenticated by subsequent blocks.

All of this works because everyone is running the same protocol using the same rule. When you

synchronize with the blockchain, you will download every block from other peers and verify that:

1. hashing a block indeed gives a digest that is smaller than some expected value.
2. the next block refers back to this block.

Not everyone has to propose blocks, but you can if you want. If you do so, you are called a **miner**. This means that in order to get your transactions in the blockchain, you need their help (as illustrated in figure 12.5).

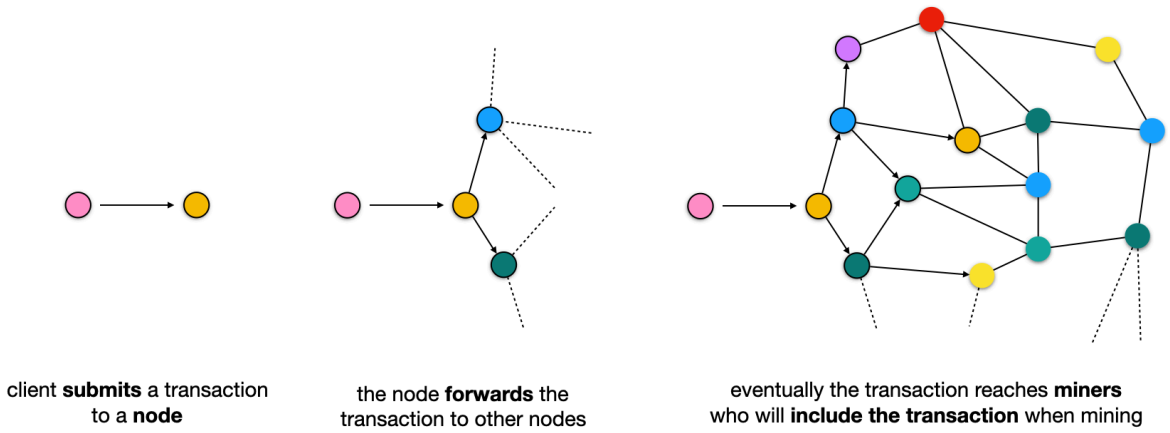


Figure 12.5 The Bitcoin network is a number of nodes (miners or not) that are interconnected. To submit a transaction, you must send it to a miner that will eventually get it into the blockchain (by including it into a block it'll mine). As you do not know who will be successful at mining the block, you must propagate your transaction to the whole network to reach as many miners as possible.

Miners do not work for free. If a miner finds a block, they collect:

- A **reward**. That is a fixed number of bitcoins that will get created and sent to your address. In the beginning, miners would get 50 BTC per block mined. But the reward value halves every 210,000 blocks, and will eventually be reduced to 0, capping the total amount of bitcoin that can be created to 21 millions.
- All the **transaction fees** contained in the block. This is why increasing the fees in your transactions allows you to get them accepted faster, as miners will tend to include transactions with higher fees in the blocks they are mining.

This is how users of Bitcoin are incentivized in making the protocol move forward. A block always contains what is called a **coinbase**, which is the address that will collect the reward and the fees. The miner usually sets the coinbase to their own address.

We can now answer the question we had at the beginning of the section: where do the first UTXOs come from? The answer is that all bitcoins in history have at some point or another been created as part of the block reward for miners.

12.2.3 Forking hell! Solving conflicts in mining

And so, Bitcoin distributes the task of choosing the next set of transactions to be processed via this proof-of-work-based system.

Your chance to mine a block is directly correlated to the amount of hash you can compute, and thus the amount of computation you can produce. A lot of computation power nowadays is directed at mining blocks in Bitcoin (or other proof-of-work-based cryptocurrencies).

> **NOTE** Proof of work can be seen as Bitcoin's way of addressing sybil attacks, which are attacks that take advantage of the fact that you can create as many accounts as you want in a protocol, giving you an asymmetric edge to dishonest participants. In Bitcoin, the only way to obtain more power is really to buy more hardware to compute hashes, not to create more addresses in the network.

There is still one problem though: the difficulty of finding a hash that is lower than some value cannot be too easy. If it is too easy, then the network will have too many participants mining a valid block at the same time. And if this happens, which mined block is the legitimate next block in the chain? This is essentially what we call a **fork**.

To solve forks, Bitcoin has two mechanisms:

Maintaining the hardness of proof of work. If blocks get mined too quickly or too slowly, the Bitcoin algorithm that everyone's running will dynamically adapt to the network conditions and increase or decrease the **difficulty** of the proof of work. Simplified, miners will have to find a block digest that has more or less zeros.

> **NOTE** If the difficulty dictates that a block digest needs to start with a 0 byte, you are expected to try 2^8 different blocks (more specifically different nonces, as explained previously) until you can find a valid digest. Raise this to 2 bytes, and you are now expected to try 2^{16} different blocks. The time it'll take for you to get there depends on the amount of power you have, and if you have specialized hardware to compute these hashes more rapidly. Currently, Bitcoin's algorithm dynamically changes the difficulty so that a block is mined every 10 minutes.

Relying on the chain with the most amount of work. The 2008 Bitcoin paper stated "the longest chain not only serves as proof of the sequence of events witnessed, but proof that it came from the largest pool of CPU power," dictating that participants should honor what they see as

the longest chain. The protocol was later updated to follow the chain with the highest cumulative amount of work (but this distinction does not matter too much here). I illustrate this in figure 12.6.

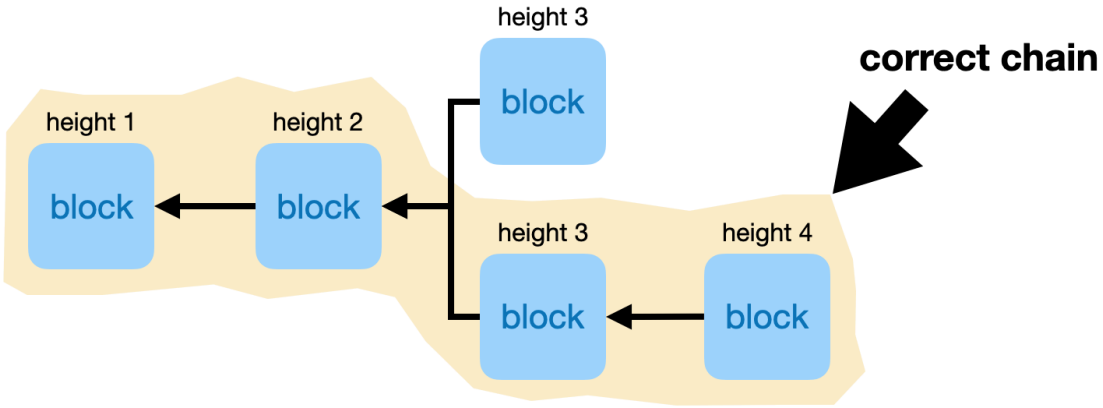


Figure 12.6 A fork in the blockchain: two miners have published a valid block at height 3 (meaning 3 blocks after genesis). Later, another miner mines a block at height 4 that points to the second block at height 3. As the second fork is now longer, it is the valid fork that miners should continue to extend. Note that arrows coming out of a block point to their parent block (the block they extend).

I've said previously that the consensus algorithm of Bitcoin is not a byzantine fault-tolerant protocol, this is because it allows such **forks**. Thus, if you are waiting for your transaction to be processed, you should absolutely not rely on observing your transaction get included in a valid block! Because the observed block could actually be a fork, and a losing one at that (to a longer fork). You need more assurance to decide when your transaction has been processed for real. Most wallets and exchange platforms will wait for a number of "confirmation" blocks to be mined on top of your block. The more blocks on top of the one that includes your transaction, the less chance that chain will be re-organized into another due to a longer existing fork.

The number of confirmation is typically set to six blocks, which makes the confirmation time for your transaction around an hour. That being said, Bitcoin still does not provide 100% assurance that a fork past 6 blocks would never happen. If the mining difficulty is well-adjusted, then it should be fine, and we have reason to believe that this is true for Bitcoin. Bitcoin's proof of work difficulty has increased gradually over time, as the cryptocurrency became more popular. The difficulty is now so high that most people cannot afford the hardware required to have a chance at mining a block. Today, most miners get together in what are called "mining pools" to distribute the work needed to mine a block. They then can share the reward.

With block 632874 [...] the expected cumulative work in the Bitcoin blockchain surpassed 2^{92} double-SHA256 hashes.

– Pieter Wuille on June 4th 2020

To understand why forks are disruptive, let's imagine the following scenario. Alice has bought a wine bottle from you, and you've been waiting for her to send you the 5 bitcoins she has in her

account. Finally, you observe a new block at height 10 (meaning 10 blocks after genesis) that includes her transaction. Being cautious, you decide to wait for 6 more blocks to be added on top of it. After waiting for a while, you finally see a block at height 16 that extends the chain containing your block at height 10. You hand out the wine bottle to Alice and call it a day.

But a few blocks later, a block at height 30 appears out of nowhere, extending a different blockchain that branched out just a block before yours (at height 9). Since the new chain is longer, it ends up being accepted by everyone as the legitimate chain. The previous chain you were on (starting from your block at height 10) gets discarded, and participants in the network simply re-organize their chain to now point to the new longest one. And as you could have guessed, this new chain doesn't include any block that includes Alice's transaction. Instead it includes a transaction moving all of her funds to another address, preventing you from republishing the original transaction that moved her funds to your address. Alice effectively **double spent** her money.

This is a **51% attack**.[28] The name comes from the amount of computation power Alice needed to perform the attack: she needed just a bit more than everyone else.

This is not just a theoretical attack, 51% attacks have happened in the real world! For example, in 2018 an attacker managed to double spend a number of funds in a 51% attack on the Vertcoin currency.

> *The attacker essentially rewrote part of the ledger's history and then, using their dominant hashing power to produce the longest chain, convinced the rest of the miners to validate this new version of the blockchain. With that, he or she could commit the ultimate crypto crime: a double-spend of prior transactions, leaving earlier payees holding invalidated coins.*
>
> *– Michael J. Casey Vertcoin's Struggle Is Real: Why the Latest Crypto 51% Attack Matters (Dec 2018)*

In 2019, the same thing happened to Ethereum classic (a variant of Ethereum), causing losses of more than $1 million at the time, with several reorganizations of more than 100 blocks of depth. In 2020, Bitcoin Gold (a variant of Bitcoin) also suffered from a 51% attack, removing 29 blocks from the cryptocurrency's history and double spending more than $70,000 in less than two days.

12.2.4 Reducing a block's size by using Merkle trees

One last interesting aspect of Bitcoin that I want to talk about is how it compresses some of the information available. A block in Bitcoin actually does not contain any transactions! Transactions are shared separately, and instead a block contains a single digest that authenticates a list of transactions. That digest could simply be the hash of all the transactions contained in the block, but it's a bit more clever than that. Instead, the digest is the root of a **Merkle tree**.

What's a Merkle tree? Simply put, it's a tree (the data structure) where internal nodes are hashes

of their children. This might be a tad confusing, and a picture is worth a thousand words, so check figure 12.7 below.

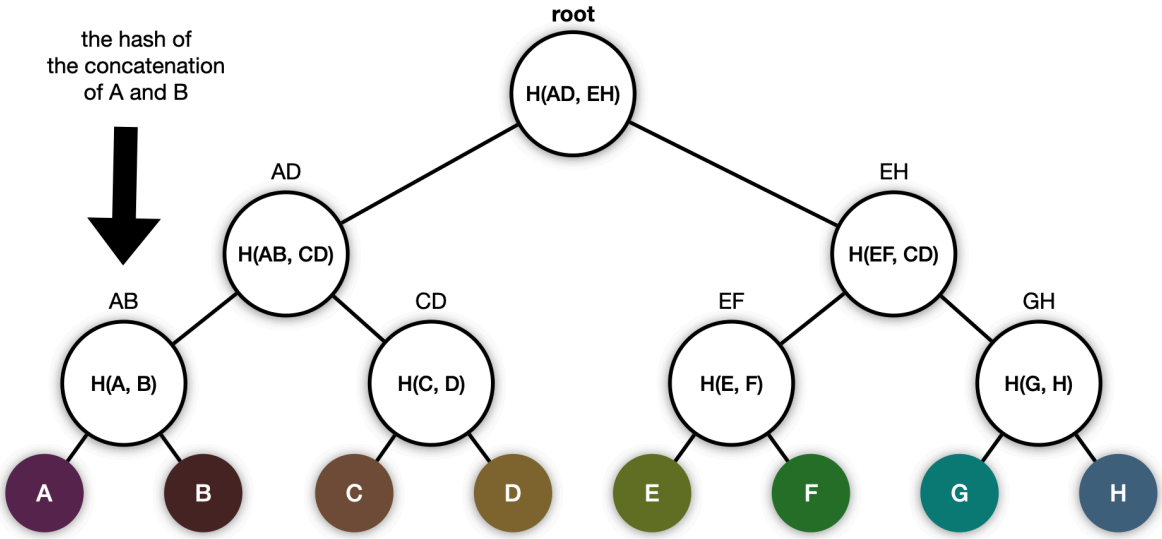
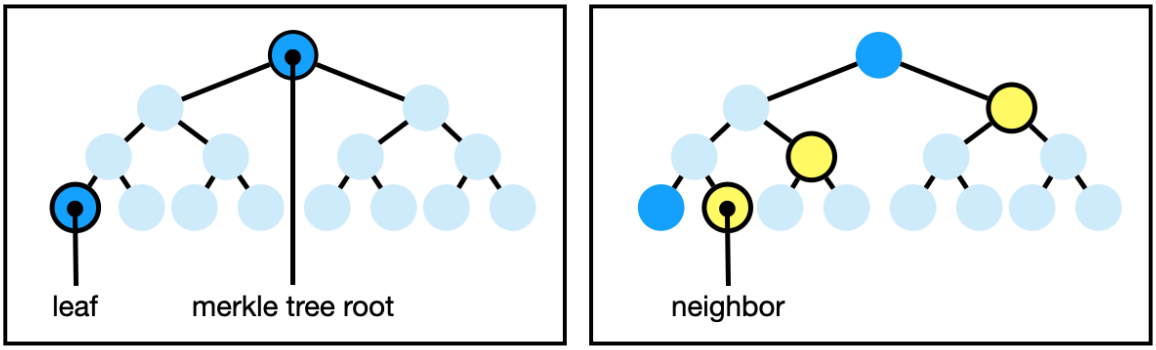


Figure 12.7 A Merkle tree, a data structure that authenticates the elements in its leaves. In the tree, an internal node is the hash of its children. The root hash can be used to authenticate the whole structure. In the diagram, `H()` represents a hash function, and the comma seperation of the inputs can be implemented as a concatenation (as long as there is no ambiguity).

Merkle trees are very useful structures and you will find them in all types of real world protocols. They can compress a large amount of data into a small fixed size value: the root of the tree.

Not only that, you do not necessarily need all the leaves to reconstruct the root. For example, imagine that you know the root of the Merkle tree due to its inclusion in a Bitcoin block, and you want to know if a transaction (a leaf in the tree) is included in the block. If it is in the tree, what I can do is to share with you the neighbor nodes in the path up to the root as a **membership proof**. (A proof that is logarithmic in the depth of the tree in size.) What's left for you is to compute the internal nodes up to the root of the tree by hashing each pair in the path. It's a bit complicated to explain this in writing, so I illustrate the proof in figure 12.8.



1. **proof of membership** for the **leaf** with knowledge of the **root**

2. using the **neighbor nodes** verify if it leads to the root

Figure 12.8 Knowing the root of a Merkle tree, one can verify that a leaf belongs to the tree by reconstructing the root hash from all the leaves. To do this, you would need all the leaves in the first place, which in our diagram is 8 digests (assuming leaves are the hashes of some object). There's a more efficient way to construct a proof of membership if you don't need all the other leaves: you only need the neighbor nodes in the path from the leaf to the root, which is 4 digests including your leaf. A verifier can then use these neighbor nodes to compute the hash of all the missing nodes in the path to the root, until they can reconstruct the root hash and see if it matches what they were expecting.

The reason for using Merkle trees in a block (instead of listing all transactions directly) is to lighten the information that needs to be downloaded in order to perform simple queries on the blockchain. For example, imagine that you want to check that your recent transaction has been included in a block, without having to download the whole history of the Bitcoin blockchain. What you can do is to only download the block headers (which are lighter as they do not contain the transactions), and once you have that, ask a peer to tell you which block included your transaction. If there is such a block, they should be able to provide you with a proof that your transaction is in the tree authenticated by the digest you have in the block header.

There's a lot more to be said about Bitcoin, but there's only so many pages. Instead, I will use the remaining pages of this book to explain how the classical Byzantine fault-tolerant consensus protocols work.

12.3 A tour of cryptocurrencies

Bitcoin is the first successful cryptocurrency, and has remained the cryptocurrency with the largest market share and value in spite of hundreds of other cryptocurrencies being created. What's interesting is that Bitcoin had, and still has, many issues that other cryptocurrencies have attempted to tackle (and some with success). Even more interesting, the cryptocurrency field has made use of many cryptographic primitives that until now did not have many practical applications, or did not even exist.

So without further ado, here is a list of issues that have been researched since the advent of Bitcoin.

Volatility. Most people currently use cryptocurrencies as speculation vehicles. The price of Bitcoin obviously helps that story, as it has shown that it can easily move thousands of dollars up or down in a single day. Some people claim that the stability will come over time, but the fact remains that Bitcoin is not usable as a currency nowadays. Other cryptocurrencies have experimented with the concept of **stablecoin**, by tying the price of their token to an existing fiat currency (like the US dollar).

Latency. There's many ways to measure how efficient a cryptocurrency is. The throughput of a cryptocurrency is the number of transactions per second that it can process. Bitcoin's throughput, for example, is quite low with only 7 transactions per second. On the other hand finality is the time it takes for your transaction to be considered finalized once it is included in the blockchain. Due to forks, Bitcoin's finality is never completely achieved, and it is considered that at least one hour after a transaction being included in a new block does the probability of it getting reverted becomes acceptable. Both numbers greatly impact the latency, which is the amount of time it takes for a transaction to be finalized from the point of view of the user. In Bitcoin this includes the creation of the transaction, the time it takes to propagate it through the network, the time it takes for it to get included in a block, and finally the hour of wait for the block to be confirmed. The solution to these speed issues can be solved by BFT protocols which usually provide finality of mere seconds with an insurance that no forks are possible, as well as throughput in the order of thousands of transactions per second. Yet, this is sometimes still not enough, and different technologies are being explored. So-called **layer 2 protocols** attempt to provide additional solutions that can enact faster payments off-chain, while saving progress periodically on the main blockchain (referred to as the layer 1 in comparison).

Blockchain size. Another common problem with Bitcoin and other cryptocurrencies is that the size of the blockchain can quickly grow to impractical sizes. This creates usability issues when users who want to use the cryptocurrency (for example to query their account's balance) are expected to first download the entire chain in order to interact with the network. BFT-based cryptocurrencies that can process a large amount of transactions per second are expected to easily reach Terabytes of data within months or even weeks. There exist several attempts at solving this. One of the most interesting ones is Mina, which doesn't require you to download the whole history of the blockchain in order to get to the latest state. Instead, Mina uses zero-knowledge proofs (mentioned in chapter 7, and that I'll cover more in depth in chapter 15) to compress all the history into a fixed-size 22KB proof. This is especially useful for lighter clients like mobile phones that usually have to trust third-party servers in order to query the blockchain.

Confidentiality. Bitcoin provides pseudo-anonymity, in that accounts are only tied to public keys. As long as nobody can tie a public key to a person, the associated account remains anonymous. Remember that all the transactions from and to that account are publicly available, and social graphs can still be created in order to understand who tends to trade more often with who, and who owns how much of the currency. There are many cryptocurrencies that attempt to

solve these issues using zero-knowledge proofs or other techniques. **Zcash** is one of the most well-known confidential cryptocurrencies, as its transactions can encrypt the sender address, receiver address, and the amount being transacted. All of that using zero-knowledge proofs.

Energy efficiency. Bitcoin has been criticized heavily for being too consuming in terms of electricity. Indeed, the university of Cambridge has recently evaluated (february 2021) that all of the energy spent mining bitcoins brings Bitcoin in the top 30 energy users in the world (if seen as a country), consuming more energy in a year than a country like Argentina. BFT protocols on the other hand do not rely on proof of work, and so avoid this heavy overhead. This is most certainly why any modern cryptocurrency seems to avoid a consensus based on proof of work, and even important PoW-based cryptocurrencies like Ethereum have announced plans to move towards greener consensus protocols.

So before going to the next chapter, let's take a look at these cryptocurrencies based on BFT consensus protocols.

12.4 DiemBFT: a byzantine fault-tolerant consensus protocol

Many modern cryptocurrencies have ditched the proof of work aspect of Bitcoin for greener and more efficient consensus protocols. Most of these consensus protocols are based on classical BFT consensus protocols that are mostly variants of the original PBFT protocol. In this last section, I will use Diem to illustrate such BFT protocols.

Diem (previously called Libra) is a digital currency initially announced by Facebook in 2019, and governed by the Diem association, an organization of companies, universities, and nonprofits looking to push for an open and global payment network. One particularity of Diem is that it is backed by real money, using a reserve of fiat currencies. This allows the digital currency to be stable, unlike its older cousin Bitcoin.

To run the payment network in a secure and open manner, a BFT consensus protocol called DiemBFT is used, which is a variant of HotStuff.

In this section, let's see how DiemBFT works.

12.4.1 Safety and liveness, the two properties of a BFT consensus protocol

A byzantine fault-tolerant consensus protocol is meant to achieve two properties, even in the presence of a tolerated percentage of malicious participants:

- **Safety**. This property states that no contradicting states can be agreed on, essentially this means that forks are not supposed to happen (or with a negligible probability).
- **Liveness**. This property states that whenever people submit transactions, the state will end up including them. In other words, nobody can stop the protocol from doing its thing.

Note that a participant is generally seen as malicious (also called **byzantine**) if they do not behave according to the protocol. This could mean that they're not doing anything, or that they're not following the steps of the protocol in the correct order, or that they're not respecting some mandatory rule meant to ensure that there is no fork, etc.

It's usually quite straightforward for BFT consensus protocols to achieve safety, while liveness is known to be more difficult. Indeed, there's a well-known impossibility result linked to BFT protocols dating from 1985,[29] which states that no **deterministic** consensus protocol can tolerate failures in an **asynchronous** network (where messages can take as many time as they want to arrive). Most BFT protocols avoid this impossibility result by considering the network somewhat **synchronous** (and indeed, no protocol is very useful if your network goes down for a long period of time) or by introducing randomness in the algorithm.

For this reason DiemBFT never forks, even under extreme network conditions. In addition it always makes progress, even when there's network partitions (where different parts of the network can't reach other parts of the network), as long as the network ends up healing and stabilizing for long enough.

12.4.2 A round in the DiemBFT protocol

Diem runs in a permissioned setting where participants—called **validators**—are known in advance. The protocol advances in strictly increasing **rounds** (round 1, 2, 3, etc.), during which validators take turns to propose **blocks** of transactions. In each round:

1. The validator that is chosen to lead (deterministically) collects a number of transactions, groups them into a new block extending the blockchain, then signs the block and sends it to all other validators.
2. Upon receiving the proposed block, other validators can vote to certify it by signing it and sending the signature to the leader of the next round.

If the leader of the next round received enough votes for that block, they can bundle all of them in what is called a **quorum certificate (QC)** (which certifies the block) and use that to propose a new block extending the now-certified block. Another way to see this is that whereas in Bitcoin a block only contains the hash of the block it extends, in DiemBFT a block also contains a number of signatures over that hash. (The number of signatures is important, but more on that later.)

Note that if validators do not see a proposal during a round (because the leader is AFK, for example), they can timeout and warn other validators that nothing happened. In this case the next round is triggered and the proposer can extend whatever is the highest certified block that they have seen. I recapitulate this in figure 12.9.

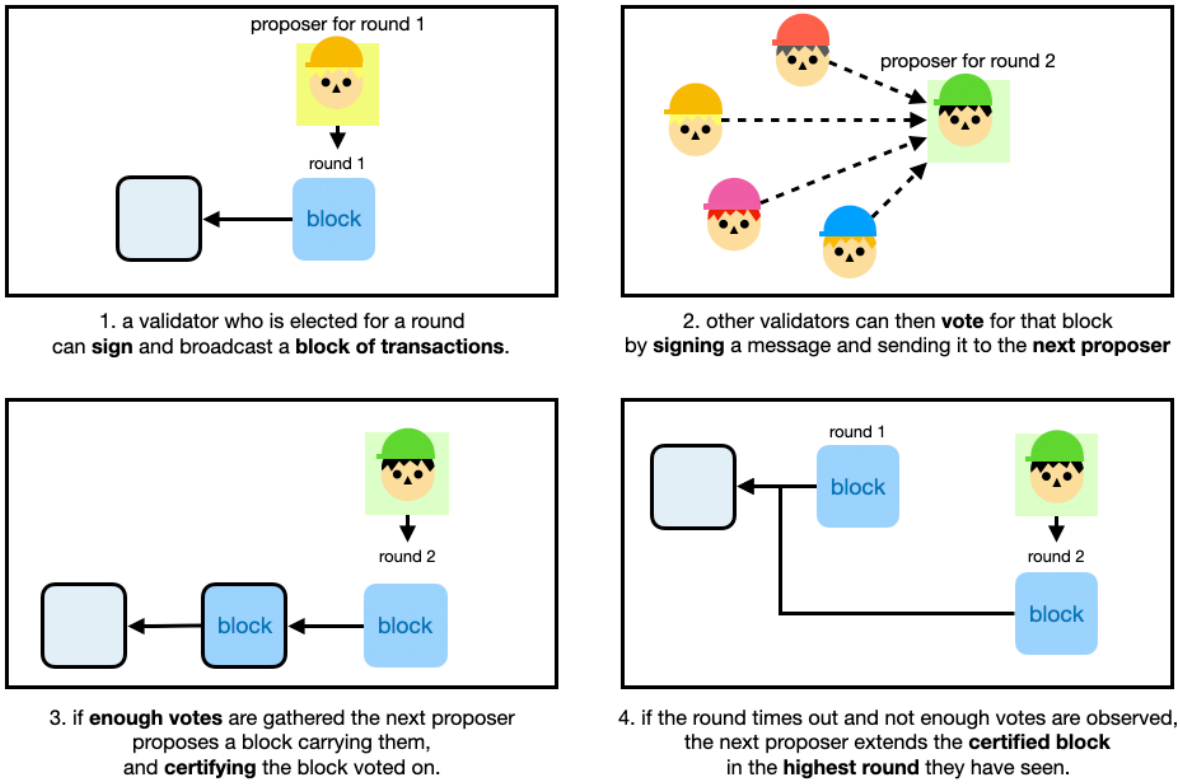


Figure 12.9 Each round of DiemBFT starts with the designated leader proposing a block that extends the last one they've seen. Other validators can then vote on this block by sending their vote to the next round's leader. If the next round's leader gathers enough votes to form a quorum certificate (QC), they can propose a new block containing the QC, effectively extending the previously seen block.

12.4.3 How much dishonesty can the protocol tolerate

Let's imagine that we want to be able to tolerate f malicious validators at most (even if they collude), then DiemBFT says that there needs to be at least $3f+1$ validators to participate in the protocol (in other words, for f malicious validators there needs to be at least $2f+1$ honest validators). As long as this assumption is true, the protocol provides safety and liveness.

With that in mind, QCs can only be formed with a majority of honest validators' votes, which is $2f+1$ signatures if there are $3f+1$ participants. These numbers can be a bit hard to visualize, so I explain how they impact confidence in the votes we observe in figure 12.10.

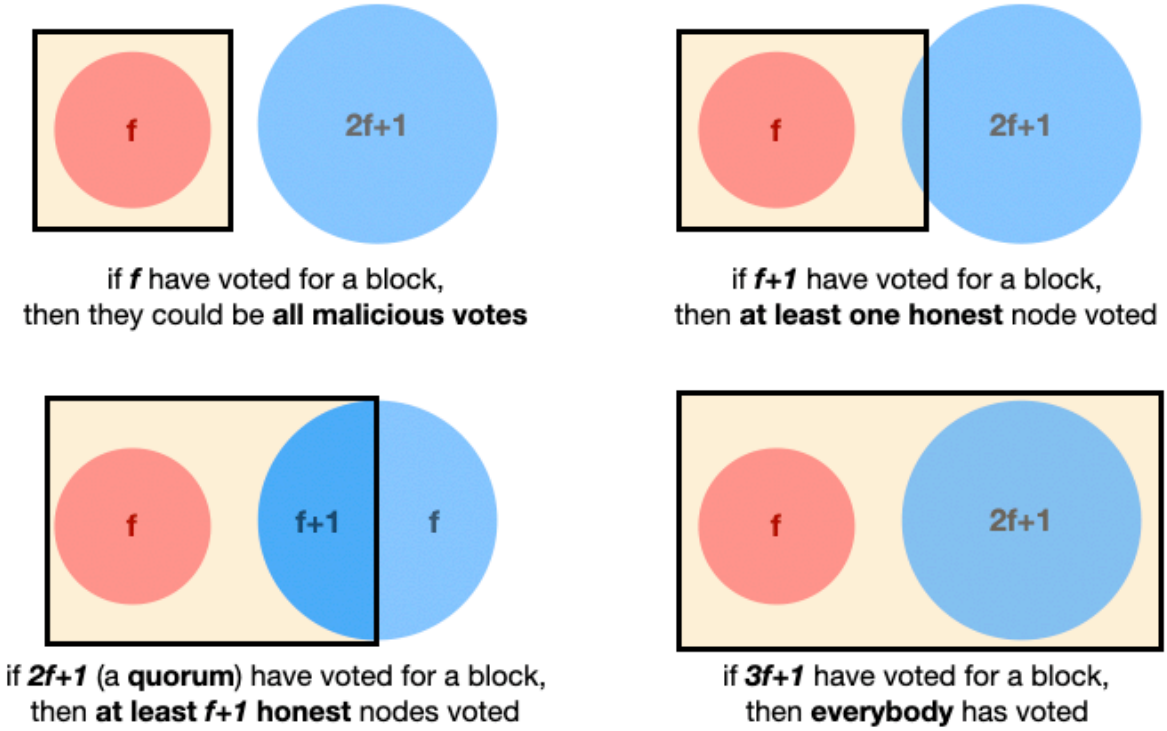


Figure 12.10 In the DiemBFT protocol, at least two thirds of the validators must be honest for the protocol to be safe (it won't fork) and live (it will make progress). In other words, the protocol can tolerate f dishonest validators if at least 2f+1 validators are honest. A certified block has received at least 2f+1 votes, as it is the lowest number of votes that can represent a majority of honest validators.

12.4.4 The DiemBFT rules of voting

Validators must follow two voting rules at all times (without which, they are considered byzantine):

1. They can't vote in the past. For example, if you just finished voting in round 3, you can then only vote in round 4 and above.
2. They can only vote for a block extending a block at their **preferred round** or higher.

What's a preferred round? By default it is 0, but if you vote for a block that extends a block that extends a block, and by that I mean you voted for a block that has a grandparent block, then that grandparent block's round becomes your preferred round (unless your previous preferred round was higher). Complicated? I know. Check figure 12.11.

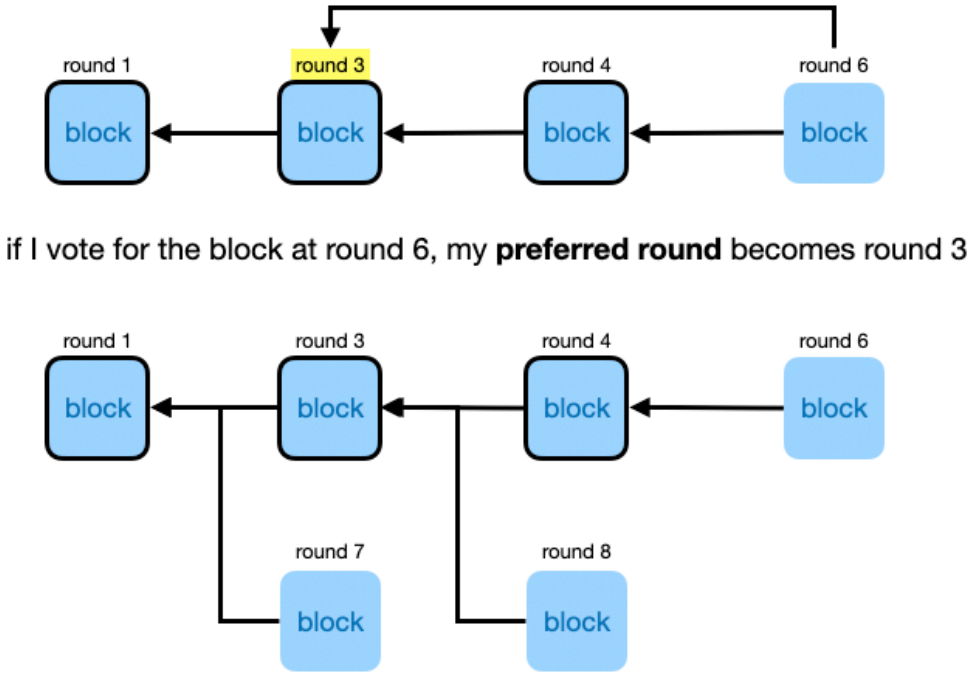


Figure 12.11 After voting for a block, a validator sets their preferred round to the round of the grandparent block if it is higher than their current preferred round. To vote on a block, its parent block must have a round greater or equal to the preferred round.

12.4.5 When are transactions considered finalized?

Note that blocks that are certified are not finalized yet, or as we also say **committed**. Nobody should assume that the transactions they contain won't be reverted.

Blocks, and the transactions they contain, can only be considered finalized once the "commit rule" is triggered. The commit rule works as follows: whenever a chain of **three blocks** is formed in **contiguous rounds** (for example, round 1, 2, and 3), the first block of the chain and the blocks it extends can get committed if the last block of the chain gets certified. Take a look figure 12.12 which shows you this in a more visual way.

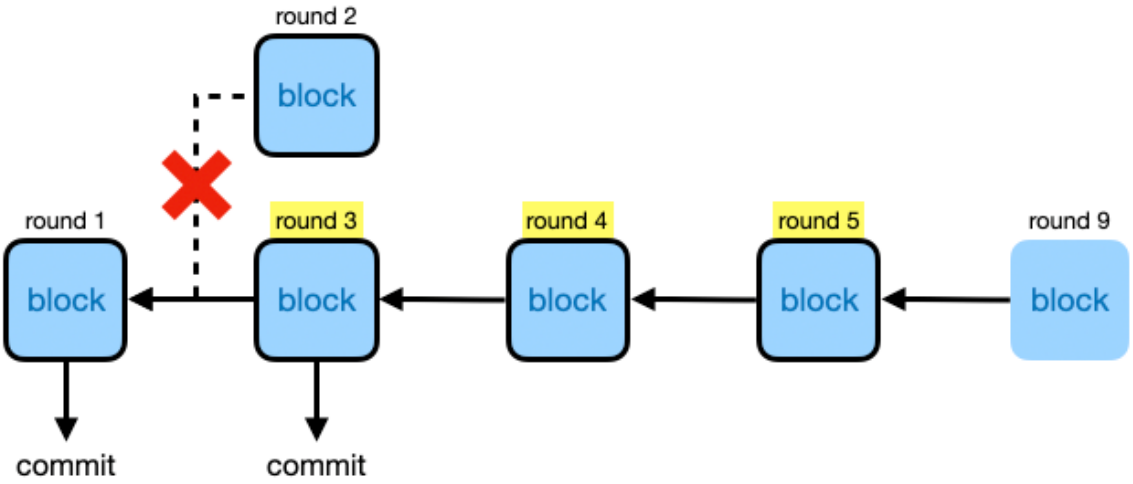


Figure 12.12 Three contiguous rounds (3, 4, 5) happen to have a chain of certified blocks. Any validator observing the certification of the last block in round 5 (by the QC of round 9) can commit the first block of the chain (at round 3), as well as all of its ancestors (here the block of round 1). Any contradicting branches (for example the block of round 2) get dropped.

And this is all there is to the protocol at a high level. But of course, once again, the devil is in the details.

12.4.6 The intuitions behind the safety of DiemBFT

While I encourage you to go read the one-page safety proof on the DiemBFT paper, I want to use a couple pages here to give you an intuition on why it works.

First, we notice that two different blocks cannot be certified during the same round. This is a very important property, which I explain more visually in figure 12.13.

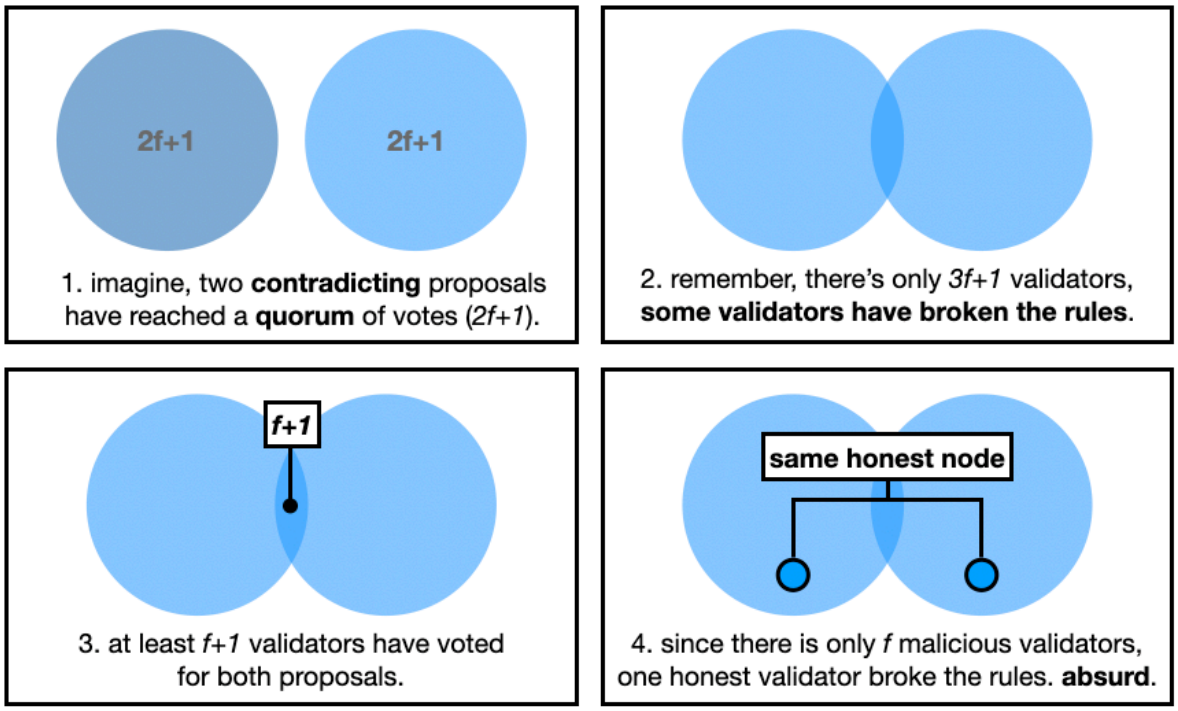


Figure 12.13 Assuming that there can only be up to f malicious validators in a protocol of 3f+1 validators, and that a quorum certificate is created from 2f+1 signed votes, then there can only be one certified block per round. The diagram shows a proof by contradiction, a proof that this cannot be, because then it would contradict our initial assumptions.

Using the property that only one block can get certified at a given round, we can simplify how we talk about blocks: block 3 is at round 3, block 6 is at round 6, and so on.

Now, take a look at figure 12.14 and take a moment to figure out why a certified block, or two certified blocks, or three certified blocks at non-contiguous rounds, cannot lead to a commit without risking a fork.

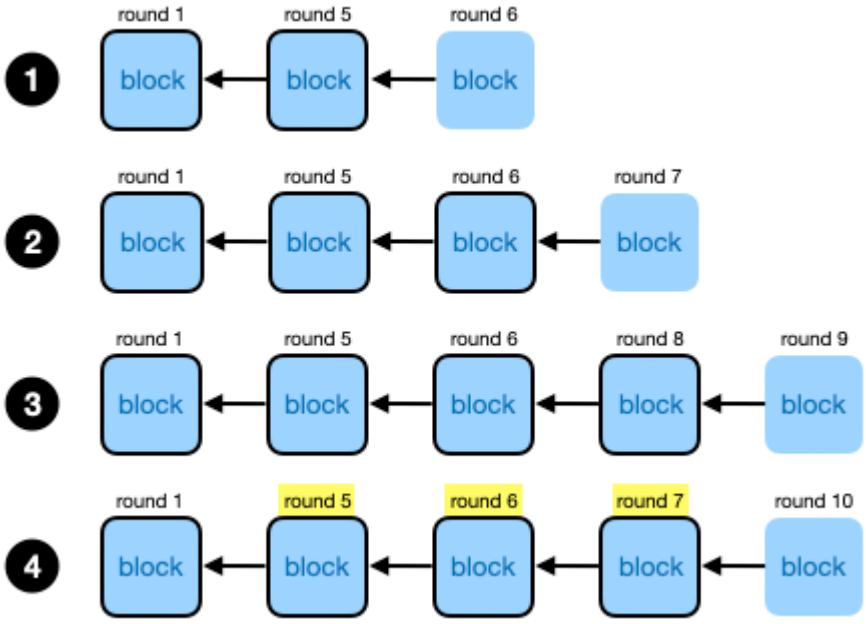


Figure 12.14 In all these scenarios, committing block 5 could lead to a fork. Only in scenario number 4, is committing block 5 safe. Can you tell why it is dangerous to commit block 5 in all scenarios but 4?

Did you manage to find out answers for all the scenarios?

The short answer is that all scenarios, for the exception of the last one, leave room for a block to extend round 1. This late block effectively branches out and can be further extended according to the rules of the consensus protocol. If this happens block 5, and other blocks extending it, will get dropped as another earlier branch gets committed.

For scenario 1 and 2, this can be due to the proposer not seeing the previous blocks. In scenario 3, an earlier block could appear later than expected, perhaps due to network delays or worse, due to a validator withholding it up to the right moment. I explain this further in figure 12.15.

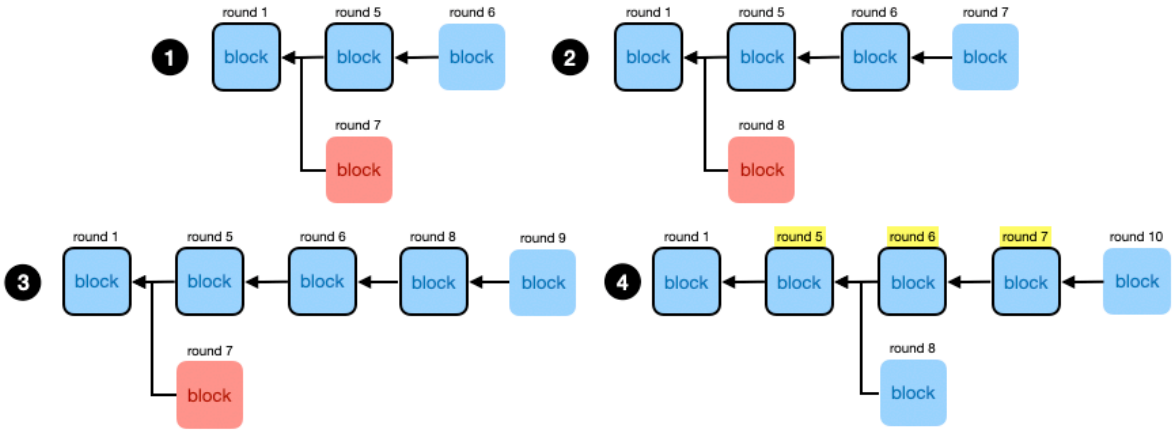


Figure 12.15 Building on figure 12.14, all scenarios except for the last one allow for a parallel chain that can eventually win and discard the branch of block 5. The last scenario has a chain of three certified blocks in contiguous rounds. This means that block 7 has had a majority of honest voters, who in turn updated their preferred round to round 5. This means that after that, no block can branch out before block 5 and obtain a QC at the same time. The worst that can happen is that a block extends block 5 or block 6, which will eventually lead to the same outcome: block 5 being committed.

12.5 Summary

- Cryptocurrencies are about decentralizing a payment network to avoid a single point of failure.
- To have everyone agree on the state of a cryptocurrency, consensus algorithms are used.
- Byzantine fault-tolerant (BFT) consensus protocols were invented in 1982 and have evolved to become faster and simpler to understand.
- BFT consensus protocols need a known and fixed set of participants to work (permissioned network). Such protocols can just decide who is part of this participant set (proof of authority) or dynamically elect the participant set based on the amount of currency they hold (proof of stake).
- Bitcoin's consensus algorithm, the Nakamoto consensus, uses proof-of-work to validate the correct chain and to allow anyone to participate (permissionless network).
- Bitcoin's proof-of-work has participants (called "miners") compute a lot of hashes in order to find some with specific prefixes. Successfully finding a valid digest allows a miner to decide on the next block of transaction, collect a reward, as well as transaction fees.
- Accounts in Bitcoin are simply ECDSA keypairs using the secp256k1 curve. An account knows how much bitcoins they hold by looking at all transaction outputs that have not yet been spent (UTXOs). A transaction is thus a signed message authorizing the movement of a number of older transaction outputs to new outputs, spendable to different public keys.
- Bitcoin uses Merkle trees to compress the size of a block and allow verification of transaction inclusion to be small in size.
- Stablecoins are cryptocurrencies that attempt to stabilize their values, most often by pegging their token to the value of a fiat currency like the US dollar.
- Cryptocurrencies use so-called layer 2 protocols in order to increase their latency by processing transactions off-chain, and saving progress on-chain periodically.
- Zero-knowledge proofs are used in many different blockchain applications, for example in Zcash to provide confidentiality, and in Coda to compress the whole blockchain to a short proof of validity.
- Diem is a stablecoin that uses a BFT consensus protocol called DiemBFT. It remains both safe (no forks) and live (progress is always made) as long as no more than f malicious participants exist out of $3f+1$ participants.
- DiemBFT works by having rounds, in which a participant proposes a block of transactions extending a previous block. Other participants can then vote for the block, potentially creating a quorum certificate if enough votes are gathered ($2f+1$).
- In DiemBFT, blocks (and their transactions) are finalized when the commit rule is triggered: a chain of 3 certified blocks at contiguous rounds. When this happens, the first block of the chain (and the blocks it extends) are committed.

13 Hardware cryptography

This chapter covers

- The issues that cryptography faces in highly-adversarial environments.
- The solutions that hardware offers to improve the attacker's cost in such environments.
- How software mitigations can also help cryptography against side-channel attacks.

At some point, writing cryptographic applications, you end up realizing that you have a number of short-term and long-term keys, and you have to make sure nobody can steal them. It means you're standing in the world of key management. Makes sense right? You've seen some of that in previous chapters, but in this chapter we'll do things a bit differently: we'll look at how key management and cryptography can be done in **highly-adversarial environments**. Environments where the attacker is much more powerful than the typical scenarios we've looked at so far.

Let's first introduce this concept in the next section. The rest of this chapter will then survey the different techniques that allow us to continue to do interesting things in spite of these constraints. Spoiler alert: it involves using specialized hardware. Finally, we'll see how cryptographic primitives have adapted to these highly-adversarial environments.

13.1 Modern cryptography attacker model

Present-day computer and network security starts with the assumption that there is a domain that we can trust. For example: if we encrypt data for transport over the internet, we generally assume the computer that's doing the encrypting is not compromised and that there's some other "endpoint" at which it can be safely decrypted.

– Joanna Rutkowska Intel x86 considered harmful (2015)

Cryptography used to be about "*Alice wants to encrypt a message to Bob, without Eve being able*

to intercept it." Today, a lot of it has moved to something more like "*Alice wants to encrypt a message to Bob, but Alice has been compromised*." It's a totally different attacker model, which is often not anticipated for in theoretical cryptography.

What do I mean by this? Let me give you some examples:

- Using your credit card on an automated teller machine (ATM) which might be augmented with a skimmer, a device that a thief can place on top of the card reader in order to copy the content of your bank card (see figure 13.1).
- Downloading an application on your mobile phone that compromises the operating system.
- Hosting a web application in a shared web hosting service, where the other customers sharing the same machine as you are malicious.
- Managing highly-sensitive secrets in a data center that gets visited by spies from a different country.

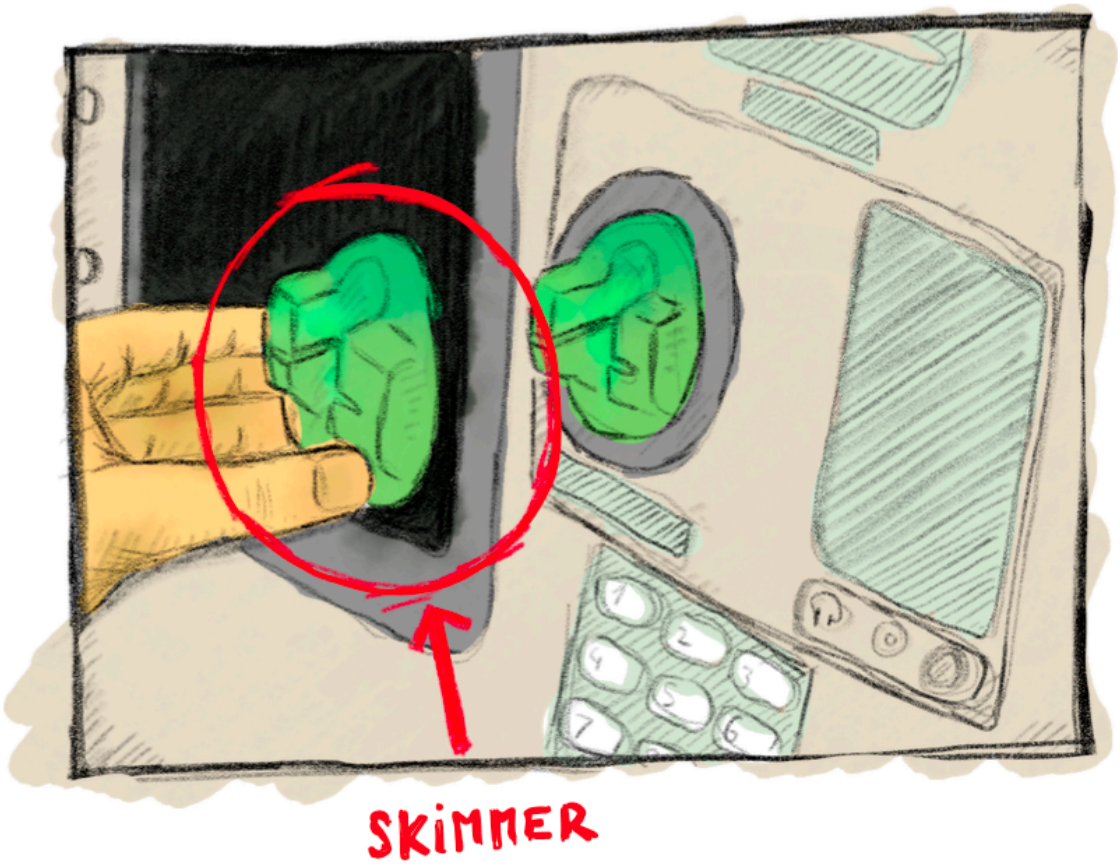


Figure 13.1 A skimmer: a malicious device that can be placed in front of an ATM card reader or a payment terminal card reader to copy data contained in the card magnetic stripe. The magnetic stripe usually contains the account number, the expiration date, and other metadata that can be used to pay online or in a number of payment terminals. Skimmers are sometimes seen accompanied with a hidden camera to obtain the user's PIN as well, potentially enabling the thief to use ATM withdrawals and payment terminals enforcing PIN entry.

All of these examples are modern use of cryptography in a threat model that many

cryptographers ignore, or are totally unaware of.

Indeed, most of the cryptographic primitives that you can read about in the literature will just assume that Alice has total control of her execution environment, and only when a ciphertext (or a signature or a public key or …) leaves her computer to go over the network, will a man-in-the-middle attacker be able to perform their tricks. But in reality, and in modern days, we're often using cryptography in much more adversarial models.

Security is, after all, a product of your assumptions and what you expect of a potential attacker. If your assumptions are wrong, you're in for a bad time.

So how do real-world applications reconcile theoretical cryptography with these more powerful attackers? It makes **compromises**. In other words, they try to make the attackers' lives more difficult. The security of such systems is often calculated in cost (how much does the attacker have to spend to break the system) rather than computational complexity.

A lot of what you'll learn in this chapter will be imperfect cryptography, which in the real-world we call **defense-in-depth**. There's a lot to learn, and this chapter comes with a lot of new acronyms and different solutions that different vendors and their marketing teams and sales people have come up with.

So let's get started and learn about **trusted systems in untrusted environments**!

13.2 Untrusted environments: hardware to the rescue

If you thought that real-world cryptography was messy and that there were too many standards or ways to do the same thing, well wait until you read what's going on in the hardware world.

There are three interesting attacker models here that we'll talk about in this chapter:

- **Software Attacks**. Your phone, your laptop, and many of your devices are not as simple as they used to be: they can now run full-featured operating systems that allow users to install applications. These systems can also communicate with the outside world via a multitude of ways (WiFi, Bluetooth, and so on). Getting a virus on your device could mean a permanent intrusion. An intrusion that could help the attacker steal your keys, or observe your cryptographic operations, or hell even replace them! Specialized hardware can help thwart such software attacks.
- **Hardware Attacks**. By default, cryptography on hardware is helpless against the threat of physical attacks. Imagine the following scenario: you leave your phone or laptop unattended in your hotel room; a malicious maid comes, opens the device, uses a low-budget off-the-shelf tool to modify the system, and then leaves the device appearing untouched where it was found before you get back to your room. Maybe he copied or changed the content of your memory (perhaps even rolling it back to a previous state), or he replaced some of the hardware, or he installed components capable of man-in-the-middle'ing some of the hardware, and so on. This is known as an **evil maid attack** in the literature and can be generalized to many situations (carrying devices in

check-in luggages while flying, storing sensitive keys in an insecure data center, and so forth). More complex and more expensive hardware attacks exist. For example hardware certification labs will often use the same tools that chip manufacturers themselves use to debug their chips. Lasers can be shot on targeted places to force a computation to produce an erroneous value (so-called **fault attacks**), chips can be opened up to reveal their parts, focused ion beam (FIB) microscopes can be used to reverse-engineer components. The sky's the limit and it is very hard to protect against such motivated attackers.

In the end, if one wants to protect against physical attacks, one typically just adds as many layers of defences as one can, in an attempt to make the attacker's life more and more difficult. **It is all about raising the costs**. In addition, it is important to design systems that do not collapse when a single instance gets broken. For cryptography this usually means that each device should use a unique key, preventing the compromise of one key to affect other devices.

NOTE **Unfortunately, the electronic world is full of bad cryptography. In 2017, the DUHK attack showed that tens of thousands of devices were using the same hardcoded seed with a random number generator, effectively breaking the security of a number of VPN appliances.**[30]

In this section we will survey different hardware solutions that attempt to protect against these different threats. As a preview you will learn about:

- **Whitebox cryptography**. A software-only mitigation that scrambles the implementation of a cryptographic primitive (like AES) with the key used, in order to prevent reverse-engineers from extracting the key.
- **Smart cards**. Small hardware chips that you can find in credit cards. They carry secret keys and perform cryptographic operations with them.
- **Secure elements**. A generalization of smart cards. The most commonly found secure elements are SIM cards in mobile phones.
- **Hardware Security Modules (HSMs)**. Larger devices that are commonly found attached to enterprise servers in data centers. Like smart cards and secure elements, HSMs also hold secret keys and perform cryptographic operations.
- **Trusted Platform Modules (TPMs)**. Repackaged secure elements or secure-element-like microcontrollers that are commonly found directly plugged into laptops and server motherboards. As you will see, many vendors invent their own TPM-like chips.
- **Hardware Security Tokens**. Keys that can are typically plugged via the USB port of a laptop and that you've already seen in chapter 11 (then called roaming authenticators).
- **Trusted Execution Environments (TEEs)**. A way to securely execute programs in a parallel environment isolated from the main operating system. In practice it is achieved via special features integrated directly into a normal processor, allowing execution of entire programs securely by the main CPU itself.

All of these solutions, for the exception of TEEs (that most often focuses on software attacks), attempt to protect againt complex physical attacks. I expect that these short descriptions made

little sense at this point, so let's take a look at each of them one by one!

13.2.1 Whitebox cryptography, a bad idea

Before getting into hardware solutions for untrusted environments, why not use software solutions? Can cryptography provide primitives that do not leak their own keys?

Whitebox cryptography is exactly this: a field of cryptography that attempts to scramble a cryptographical implementation with the key it uses, in order to prevent extraction of that key. That's right, the attacker can be given the source code of some whitebox AES-based encryption algorithm with a fixed key, and it will encrypt and decrypt fine, but the key is mixed so well with the implementation that it will be too hard for anyone to extract it from the algorithm. That's the theory at least. In practice, no published whitebox crypto algorithm has been found to be secure, and most commercial solutions are closed-source due to this fact (security through obscurity kinda works in the real-world, but should generally not be relied upon).[31] Again, it's all about raising the cost and making it harder for attackers.

This type of technique (scrambling an implementation to make it hard to understand) is called **obfuscation** in security, and it is rarely seen as a good way to provide strong security. Nonetheless it is not a useless technique! Remember, defense-in-depth is a thing in the real-world, and slowing down attackers is sometimes a desirable property of a system.

All in all, whitebox cryptography is a big industry that sells dubious products to businesses in need of Digital Rights Management (DRM) solutions.

> **NOTE** **Briefly, DRM is a tool that controls how much access a customer can get to a product they bought. For example, you can find DRM in hardware that can play movies you bought in a store, or in software that can play movies you are watching on a streaming service. In reality DRM does not strongly prevent these "attacks," it just makes the life of their customers more difficult.**

On the more serious side, there is a branch of cryptography called **Indistinguishability obfuscation (iO)** that attempts to do this cryptographically (so for realz). iO is a very theoretical, impractical, and so far not-a-really-proven field of research. We'll see how that one goes, but I wouldn't hold my breath.

13.2.2 You probably have one in your wallet: smart cards

OK, whitebox cryptography is not great, and that's pretty much the best software can do to defend against powerful adversaries. So let's turn to the hardware side for solutions, spoiler alert: things are about to get much more complicated and confusing. Different terms have been made up and used in different ways, and standards have unfortunately proliferated as much as (if not more than) cryptographic standards.

To understand what all of these hardware solutions are, and how they differ from one another, let's start with some necessary history.

Smart cards are small chips usually seen packaged inside plastic cards like bank cards, that is if your bank follows the EMV standard,[32] and were invented in the early 70s following advances in microelectronics. Smart cards started as a practical way to get everyone a **pocket computer**! Indeed, a modern smart card embeds its own CPU, different types of programmable or non-programmable memory (ROM, RAM, and EEPROM), inputs and outputs, a hardware random number generator (also called TRNG as you've learned in chapter 8), and so on. They're "smart" in the sense that they can run programs, unlike the not-so-smart cards that could only store data via a magnetic stripe (which could be easily copied via the skimmers I talked about previously). Most smart cards allow developers to write small and contained applications that can run on the card. The most popular standard supported by smart cards is JavaCard, which allow developers to write Java-like applications.

Today, it seems like most people have a much more powerful computer in their pockets, so smart cards are probably going to die.

Smart cards need to be activated by inserting it into a card reader. More recently cards have been augmented with the near-field communication (NFC) protocol to achieve the same result via radio frequencies (so by getting close to, as opposed to touching, another device).

> **NOTE** By the way, banks make use of smart cards to store a unique per-card secret capable of saying "I am indeed the card that you gave to this customer". Intuitively, you might think that this is implemented via public-key cryptography, but the banking industry is still stuck in the past and uses symmetric cryptography (due to the vast amount of legacy software and hardware still in use)! More specifically, every bank card stores a triple-DES (also called 3DES) symmetric key, an old 64-bit block cipher that improves on the Data Standard Encryption (DES) which was deprecated in favor of AES (covered in chapter 4). The algorithm is used not to encrypt, but to produce a MAC over some challenge. The MAC can be verified by the bank who holds every customer's current 3DES symmetric key. This is an excellent example of what real-world cryptography is often about: legacy algorithms used all over the place in a risky way. And this is also why key rotation is such an important concept (and why you have to change your bank cards so often).

Smart cards mix a number of physical and logical techniques to prevent observation, extraction, and modification of its execution environment and some of its memory (where secrets are stored). But as I said earlier, it's all about how much money an attacker is willing to spend. There exist many attacks that attempt to break these cards, which can be classified in three different categories:

- **Non-invasive attacks** such as differential power analysis (DPA) which analyze the power consumption of a smart card while it is doing cryptographic operations in order to extract its associated keys. Power consumption is not the only way cryptographic operations can leak critical information, as you will see later in this chapter in the section on side-channel attacks. (Although you've already seen how time can negatively affect cryptographic algorithms in chapter 3.)
- **Semi-invasive attacks** can use access to the chip's surface to mount attacks such as differential fault analysis (DFA) which use heat, lasers, and other techniques to modify the execution of a program running on the smart card in order to leak keys via cryptographic attacks.
- **Invasive attacks** can modify the circuitry in the silicon itself to alter its function and reveal secrets. These attacks are noticeable (because the device is damaged afterwards) and have higher chances of rendering the device unusable.

The fact that these hardware chips are extremely small and tightly packaged can make attacks very difficult, but specialized hardware usually go much further by using different layers of materials to prevent depackaging and physical observation, and by using hardware techniques to increase the innacuracy of known attacks.

For example, the ATECC508A chip claims the following hardware protections:

> *Physical and cryptographic countermeasures make it impossible for an attacker to sniff operations to learn the keys, or probe the device to obtain the keys. The entire device is shielded with a serpentine metal pattern that prevents internal signal emissions from being detectable outside and provides a visual barrier against someone opening the package to observe and probe operations. The shield is electrically connected to the rest of the circuit. If it is compromised, the device will no longer operate, preventing a determined attacker from probing circuit nodes to learn the secrets. Regulators and counters are used to confound power and signal signatures. There are no extra internal pads for test and debug, so opening the package provides no additional access points.*
>
> *– Atmel: Security for Intelligent Connected IoT Edge Nodes*

Software can also help thwart some physical attacks, for example by adding jitter to the power consumption, or the clock. But more interestingly, cryptographic implementations can be implemented in ways that can render some of these attacks impossible. In the last section of this chapter, I will spend some time going over these implementation techniques.

13.2.3 Secure elements: a generalization of smart cards

Smart cards got really popular really fast, and it became obvious that having such a secure blackbox in other devices could be useful. The concept of a **secure element** was born: a tamper-resistant microcontroller that can be found in a plugxgable form factor like UUICs (also known as SIM cards, which are required by carriers for your phone to access their network) or directly bonded on chips and motherboards like the embedded SE (eSE) attached to an iPhone's NFC chip. A secure element is really just a small **separate** piece of hardware meant to protect your secrets and their usage in cryptographic operations.

Secure elements are an important concept to protect cryptographic operations in the **Internet of Things (IoTs)**, a colloquial (and overloaded) term to refer to devices that can communicate with other devices (think smart cards in credit cards, SIM cards in phones, biometric data in passports, garage keys, smart home sensors, and so on).

Thus, you can see all of the solutions that will follow in this section as secure elements implemented in different form factors, using different techniques to achieve the same thing, but providing different levels of speed as well as defenses against software and hardware attackers.

The main definitions and standards around secure elements have been produced by Global Platform, a nonprofit association created from the need of the different players in the industry to facilitate interoperability between different vendors and systems. There exist more standards that focus on the security claims of secure elements like Common Criteria (CC), NIST's FIPS, EMV (for Europay, Mastercard, and Visa), and so on. If you're in the market for buying secure microcontrollers, you will often see claims like "FIPS 140-2 certified" and "certified CC EAL 5+" in the product's description. These are claims that can be obtained after spending some quality time, and a lot of money, with licensed certification labs.

As secure elements are highly secretive recipes, integrating them in your product means that you will have to sign non-disclosure agreements and use closed-source hardware and firmware. For many projects, this is seen as a serious limitation in transparency, but can be understood as part of the security in these chips come from the obscurity of their design.[33]

Next, let's look at hardware security tokens!

13.2.4 Enforcing user intent with hardware security tokens

Hardware security tokens are keys that you can usually plug into your machine and that can do some cryptographic operations. I've mentioned hardware tokens and how they work in chapter 11. If you remember, YubiKeys are small dongles that you can plug in the USB port of a laptop, and that will perform some cryptographic operations if you touch their exposed yellow rings.[34]

It's not clear how much protection against hardware attacks your typical hardware security token

has to implement, since they are usually used in addition to a second authentication factor like a password. Thus, the compromise of a security key is not enough to successfully authenticate as a user (unless of course it is used as single factor authentication). Yet, certifications like FIDO's "Authenticator Certification Levels" exist, and higher-levels mandate the use of specialized hardware; thus YubiKeys typically have a secure element inside. Still, this doesn't exclude software attacks if badly programmed.[35]

Cryptocurrencies have similar dongles called hardware wallets that can store account keys, and sign transactions for a user. For these cryptocurrency hardware wallets, the threat model is different; all of the digital funds' security is reduced to the security of the key stored in these devices. Thus, such hardware dongles will usually implement another layer of authentication (for example by having the user manually enter a PIN on the device).

Let's look at TPMs next.

13.2.5 Trusted Platform Modules (TPMs): a useful standardization of secure elements

Secure elements are useful, but they are either quite limited to some specific use cases, or one has to write custom applications in order to create new use cases. For this reason the **Trusted Computing Group (TCG)**, another nonprofit organization formed from different industry players, published the first **Trusted Platform Module** standard in 2009. The latest version of the standard is TPM 2.0, published as the International Organization for Standardization and the International Electrotechnical Commission (ISO/IEC) 11889.

A TPM complying with the TPM 2.0 standard is a secure microcontroller that carries a hardware random number generator, secure memory for storing secrets, can perform cryptographic operations, and the whole thing is tamper resistant. If this description reminds you of smart cards and secure elements, I told you that everything you were going to learn about in this chapter was going to be secure elements of some form or another. And actually, it's common to see TPMs implemented as a repackaging of secure elements.

You usually find a TPM directly soldered or plugged to the motherboard of enterprise servers, laptops, and desktop computers (see figure 13.2).

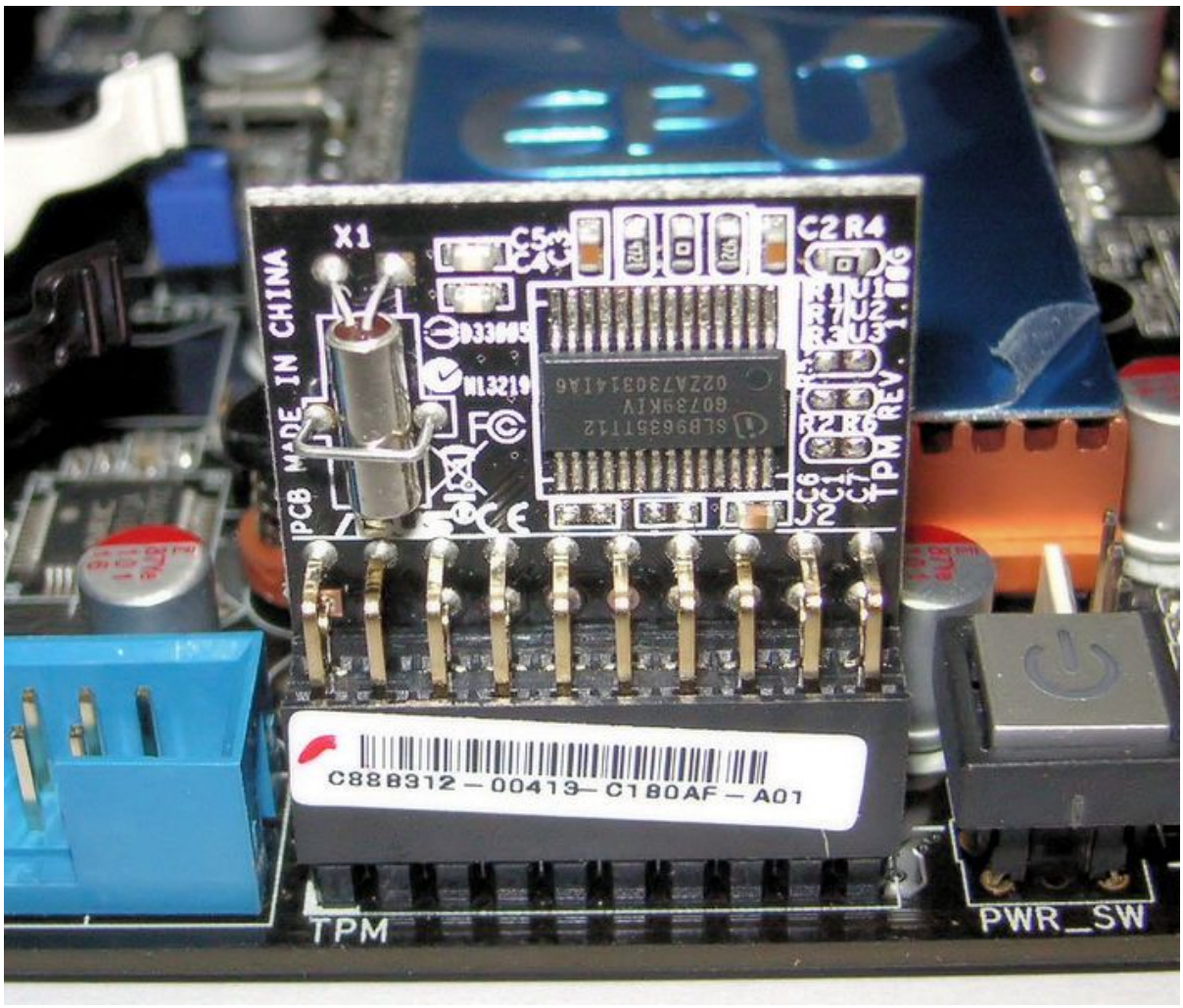


Figure 13.2 A chip implementing the TPM 2.0 standard, plugged into a PC's motherboard. This chip can be called by motherboard components of the system as well as user-applications running on the PC's operating system.

Unlike smart cards and secure elements, a TPM does not run arbitrary code, but offers a well-defined interface that a greater system can take advantage of. TPMs are usually pretty cheap, allowing more and more IoTs to ship with one.

Here is a non-exhaustive list of interesting applications that a TPM can enable:

- **User authentication**. TPMs can be used to verify a user PIN or password that is stored in the TPM itself. After that, the TPM can release secrets or perform cryptographic operations with its unique per-TPM key (called an endorsement key) to authenticate the user to a remote service (for example by signing a random challenge). In order to prevent low entropy credentials to be easily brute-forced, a TPM can also rate limit or even count the number of failed attempts. More clever approaches limit brute-force attacks by using a password-based key derivation function in order to verify the credentials (see chapter 2).
- **Measured boot**. Measured boot is about measuring what is being launched at boot. For this, the TPM offers special "registers" (called Platform Configuration Registers) that reset only when power is shut down. These registers are "rolling" hashes, meaning that

you can only extend them and not replace them (they can only hash the previous hash with new input). During booting, the first piece of code ran will hash the image of the next piece of code that will be booted and send it to the TPM. The second piece of code will do the same with every piece of code that it will boot next. And so on.

- **Full disk encryption (FDE)**. FDE encrypts the entire content of a disk with a key stored in the TPM in case your device becomes lost or stolen. A TPM can enforce user authentication and ensures that the system has reached a known good state (via measured boot) before releasing the decryption key. When the device is locked or shut down, the key vanishes from memory and has to be released by the TPM again. Note that the TPM has to release the key because it is too slow to decrypt disk data on the fly.
- **Remote attestation**. This allows a device to prove to a remote service that it is running specific software. For example, during employee onboarding a company can add a new employee's laptop's TPM endorsement key to a whitelist of approved devices. Later, if the employee wants to access one of the company's services, the service can request the employee's TPM to sign a random challenge (to prevent replays) along with the result of the measured boot to authenticate the employee and prove the well-being of their device.
- **Secure boot**. Secure boot is stronger than measured boot: it is about enforcing that the system starts in a good state in order to avoid tampering of the OS by malware (or light physical intrusions). During boot, the startup code initializes the measuring registers in the TPM, then hashes the code of the next image to load and asks the TPM to compare these with the expected hashes before actually running the code. The process will fail to boot if the TPM cannot successfully match the hash of that next piece of code. If you hold a public key you can also verify that a piece of code has been signed before running it. Note that the very first piece of code booted (usually called the Core Root of Trust for Measurement), or the very first public key used to verify that code, is the **root of trust**. That root of trust is thus usually stored in read-only memory, fused during manufacturing, and cannot be modified.

> **NOTE** Interestingly, the Nintendo switch was found to have a bug in its bootrom, which is the first piece of code that boots and as its name indicates is stored in read-only memory. Because of that, the bug is unpatchable.

These are only a few functionalities that a TPM can enable, and there are many more of them. There's after all hundreds of commands in the TPM standards.

Now the bad: the communication channel between a TPM and a processor can be man-in-the-middled easily.[36] While many TPMs provide a high level of resistance against physical attacks, the fact that their communication channel is somewhat open does reduce their use cases to mostly defend against software attacks. To solve these issues, there's been a move to TPM-like chips that are integrated directly into the main processor. For example Apple has the Secure Enclave, Microsoft has Pluton, and Google has Titan.

13.2.6 Banks love them: hardware security modules (HSMs)

If you understood what a secure element was, well a **hardware security module (HSM)** is pretty much a **bigger and faster** secure element. It's not always true, some are small,[37] and the term hardware security module can be used to mean different things by different people. Many would argue that all of the hardware solutions discussed so far are HSMs of different form factors, and that secure elements are just HSMs that are specified by GlobalPlatform while TPMs are HSMs that are specified by the Trusted Computing Group. But most of the time, when people talk about HSMs, they mean the big stuff. Like some secure elements, some HSMs can run arbitrary code as well. HSMs typically contain specialized hardware designed to perform certain cryptographic operations faster than a standard general-purpose computer, thus providing some degree of crypto acceleration.

HSMs are often classified according to FIPS 140-2: Security Requirements for Cryptographic Modules. The document is quite old, published in 2001, and does not take into account a number of attacks discovered after its publication.[38] Fortunately, it is superseded by the more modern version FIPS 140-3, and will be phased out by the end of 2021. FIPS 140 defines security levels for hardware security modules between 1 and 4; level 1 HSMs do not provide any protection against physical attacks, while level 4 HSMs will wipe their secrets if they detect any intrusion!

> **NOTE** **Wiping secret is a practice called zeroization. Some HSMs overwrite the data multiple times, even if unpowered thanks to backup internal batteries. And that's why there's very few level 4 HSMs in the market.**

The US, Canada, and some other countries mandate certain industries like banks to use devices that have been certified according to the FIPS 140 levels. Many companies in the world have decided to follow these same recommendations.

Typically, you find an HSM as an external device with its own shelf on a rack (see figure 13.3) plugged to an enterprise server in a data center (), as a PCIe card plugged into a server's motherboard, or even as small dongles that resemble a hardware security token and that you can plug via USB (if you don't care about performance).

IBM 4767 cryptographic adapter (HSM) - picture form Eigenes Werk

Figure 13.3 An IBM 4767 HSM as a PCI card.

To go full circle, some of these HSMs can be administered using smart cards (to install applications, backup keys, and so on).

HSMs are highly used in some industries. Every time you enter your PIN in an ATM or a payment terminal, the PIN ends up being verified by an HSM somewhere. Whenever you connect to a website via HTTPS, the root of trust comes from a Certificate Authority (CA) that stores its private key in an HSM, and the TLS connection is possibly terminated by an HSM. You have an Android or iPhone? Chances are that Google, or Apple, are keeping a backup of your phone safe with a fleet of HSMs. This last case is interesting because the threat model is reversed: the user does not trust the cloud with its data, and thus the cloud service provider claims that its service can't see the user's encrypted backup nor can access the keys used to encrypt it.

HSMs don't really have a standard, but most of them will at least implement the **Public-Key Cryptography Standard 11 (PKCS#11)**, one of these old standards that were started by the RSA company and that were progressively moved to the OASIS organization (2012) in order to facilitate adoption of the standards.

While the last version (2.40) of PKCS#11 was released in 2015, it is merely an update of a standard that originally started in 1994. For this reason it specifies a number of old cryptographic algorithms, or old ways of doing things, which can of course lead to vulnerabilities.[39] Nevertheless, it is good enough for many uses, and specifies an interface that allows different systems to easily interoperate with each other. The good news is that PKCS#11 v3.0 might be released at the time this writing gets printed in ink, which will include a lot of modern cryptographic algorithms like Curve25519, EdDSA, and SHAKE to name a few.

While HSMs' real goal is to make sure nobody can extract key material from them, their security

is not always shining. A lot about the security of these hardware solutions really relies on their high price, the protection techniques used not being disclosed, and the certifications (like FIPS and Common Criteria) mostly focusing on the hardware side of things. In practice, devastating software bugs have been found and it is not always straightforward to know if the HSM you use is vulnerable to any of these vulnerabilities.[40]

> **NOTE** **Not only the price of one HSM is high (it can easily be tens of thousands of dollars depending on the security level), in addition to an HSM you often have at least another HSM you use for testing, and at least another HSM you use for backup (in case your first HSM dies with its keys in it). It can add up!**

Furthermore, I still haven't touched on the elephant in the room with all of these solutions: while you might prevent most attackers from reaching your secret keys, you can't prevent attackers from compromising the system and making their own calls to the secure hardware module (be it a secure element or an HSM). Again, these hardware solutions are not a panacea and depending on the scenario they provide more or less defense-in-depth.

13.2.7 Modern integrated solutions: Trusted Execution Environment (TEE)

So far, all of the hardware solutions we've talked about have been **standalone** secure hardware solutions (with the exceptions of smart cards which can be seen as tiny computers). Secure elements, HSMs, and TPMs can be seen as an additional computer.

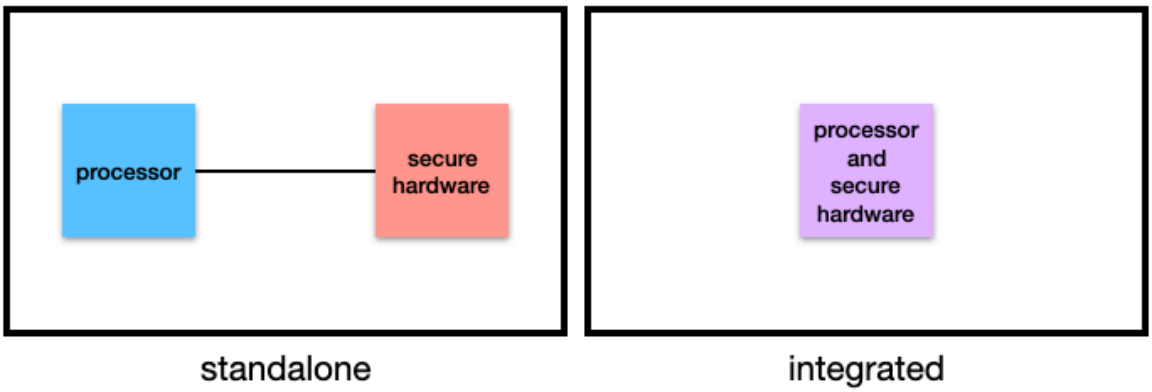


Figure 13.4 Standalone security modules typically carry their own CPU, memory, true random number generator, and so on, and talk to the main processor via system bus or other wires. Integrated security modules on the other hand are implemented directly in the main processor and thus benefits from a lot of its features and speed.

Let's now talk about **integrated** secure hardware!

Trusted Execution Environment (TEE) is a concept that extends the instruction set of a processor to allow for programs to run in a separate secure environment. The separation between this secure environment and the ones we are used to dealing with already (often called "rich"

execution environment) is done via hardware. So what ends up happening is that modern CPUs run both a normal OS as well as a secure OS simultaneously. Both have their own set of registers but share most of the rest of the CPU architecture. By using CPU-enforced logic, data from the secure world cannot be accessed from the normal world. For example a CPU will usually split its memory, giving one part for the exlusive use of the TEE. The TEE also has access to a read-only key (usually fused manually at manufacturing) that it can use to wrap other keys, or sign payloads from the TEE (attestations). Due to TEE being implemented directly on the main processor, not only does it mean a TEE is a faster and cheaper product than a TPM or secure element, it also comes for free in a lot of modern CPUs.

TEE like all other hardware solutions has been a concept developed independently by different vendors, and then a standard trying to play catch up (by Global Platform). The most known TEEs are Intel's **Software Guard Extensions (SGX)** and ARM's **TrustZone**. But there are many more like AMD PSP, RISC-V MultiZone and IBM Secure Service Container.

By design, since a TEE runs on the main CPU and can run any code given to it (in a separate environment called an "enclave" or "secure world"), it offers more functionality than secure elements, HSMs, TPMs (and TPM-like chips). For this reason TEEs are used in a wider range of applications in cloud services when clients don't want to trust the servers with their own data.

TEE's goal is to first and foremost thwart **software attacks**. While the claimed software security seems to be really attractive, it is in practice hard to segregate execution while on the same chip due to the extreme complexity of modern CPUs and their dynamic states, as can attest to the many software attacks against SGX and TrustZone.[414243444546474849]

In theory, a TPM can be reimplemented in software only via a TEE, but one must be careful as again, TEE as a concept provides no resistance against hardware attacks besides the fact that things at this microscopic level are way too tiny and tightly packaged together to analyze without expensive equipment. But by default a TEE does not mean you'll have a secure internal storage (you need to have an additional fused key that can't be read to encrypt what you want to store), or a hardware random number generator, and other wished hardware features. But every manufacturer sure has different offers with different levels of physical security and tamper resistance when it comes to chips that support TEE.

13.3 What solution is good for me?

You have learned about many hardware products in this chapter. As a recap, this is the list:

- **Smart cards** are microcomputers that need to be turned on by an external device like a payment terminal. They can run small Java-like applications. Bank cards are an example of a widely used smart card.
- **Secure elements** are a generalization of smart cards, which rely on a set of Global Platform standards. SIM Cards are secure elements for example.

- **TPMs** are repackaged secure elements plugged on personal and enterprise computers' motherboards. They follow a standardized API (by the Trusted Computing Group) that are used in a multitude of ways from measured/secure boot with FDE to remote attestation.
- **Hardware Security Tokens** are dongles like YubiKeys that often repackage secure elements to provide a 2nd factor by implementing some authentication protocol (usually TOTP or FIDO2).
- **HSMs** can be seen as external and big secure elements for servers. They're faster and more flexible. Seen mostly in data centers to store keys.
- **TEEs** like TrustZone and SGX can be thought of as secure elements implemented within the CPU. They are faster and cheaper but mostly provide resistance against software attacks unless augmented to be tamper-resistant. Most modern CPUs ship with TEEs and various levels of defense against hardware attacks.

I illustrate all these in figure 2.19.

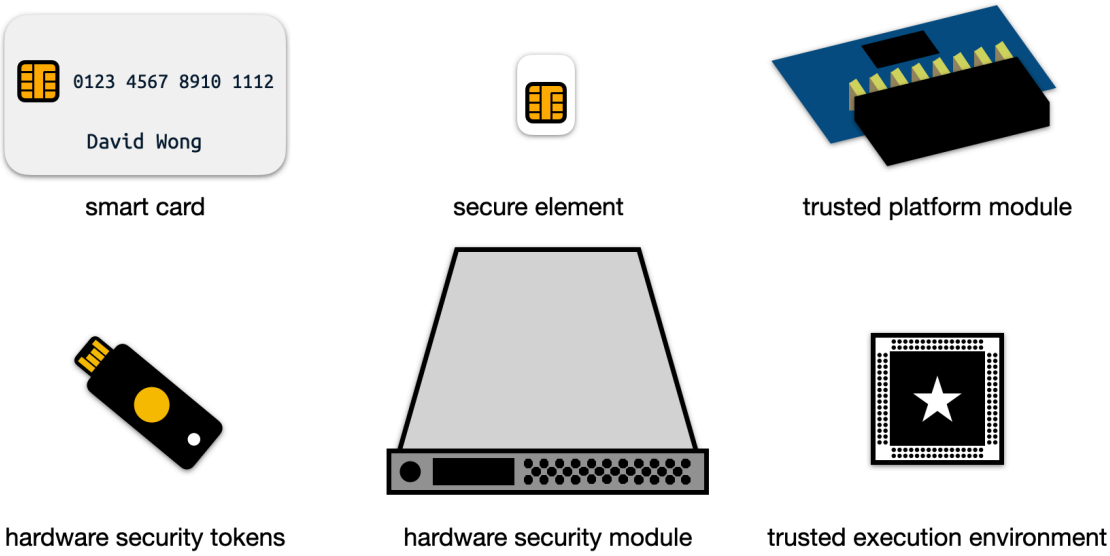


Figure 13.5 The different hardware solutions you've learned in this chapter and an idea of what they look like.

What is the best solution for you? Try to narrow your choice by asking yourself some questions:

- In what form factor? For example, the need for a secure element in a small device will dictate what solutions you won't be able to use.
- How much speed do you need? Applications that need to perform a high number of cryptographic operations per second will be highly constrained in the solutions they can use: probably limited to HSMs and TEEs.
- How much security do you need? Certifications and claims by vendors will correspond to different levels of software or hardware security. The sky's the limit.

Keep in mind that no hardware solution is the panacea, you're only increasing the attack's cost. Against a sophisticated attacker all of that is pretty much useless. Design your system so that one device compromised doesn't imply all devices are compromised. Even against normal

adversaries, compromising the main operating system often means that you can make arbitrary calls to the secure element. Design your protocol to make sure that the secure element doesn't have to trust the caller by either verifying queries, or relying on an external trusted part, or by relying on a trusted remote party, or by being self-contained, and so on. And after all of that, you still have to worry about side-channel attacks :)

After all of that, you still have more problems: how do you know all of this was done correctly? How do you know that no backdoor was inserted in the hardware you're relying on at some point during manufacturing or transport (so-called **supply chain attacks**)?

It's turtles all the way down. So this is where I'll stop.

13.4 Leakage-resilient cryptography - or how to mitigate side-channel attacks in software

We've seen how hardware attempts to prevent direct observation and extraction of secret keys, but there's only so much the hardware can do. At the end of the day, it is possible for the software to not care and give out the key despite all of this hardware hardening. The software can do so somewhat directly (like a backdoor) or it can do it indirectly by leaking enough information for someone to reconstruct the key. This latter option is called a side-channel, and side-channel vulnerabilities are most of the time non-intended bugs (at least one would hope).

I've mentioned **timing attacks** in chapter 3, where you learned that MAC authentication tags had to be compared in constant-time otherwise attackers could infer the correct tag after sending you many incorrect ones and measuring how long they waited for you to respond. Timing attacks are usually taken seriously in all areas of real-world cryptography due to the fact that they can potentially be remotely exploited over the network unlike physical side-channels.

The most important and known side-channel is **power consumption** which I've mentioned earlier in this chapter. It was discovered as an attack called **Differential Power Analysis (DPA)** by Kocher, Jaffe, and Jun in 1998; when they realized that they could hook an oscilloscope to a device, and observe variance in the electricity consumed by the device over time when performing encryption of known plaintexts. This variance clearly dependend on the bits of the key used, and the fact that operations like XORing would consume more or less power depending if the operand bits were set or not. This observation led to a key-extraction attack (so-called "total breaks").

This concept can be illustrated with **Simple Power Analysis (SPA)** attacks. In some ideal situations, and when no hardware or software mitigations are implemented against power analysis attacks, it suffices to measure and analyze the power consumption of a single cryptographic operation involving a secret key. I illustrate this in figure 13.6.

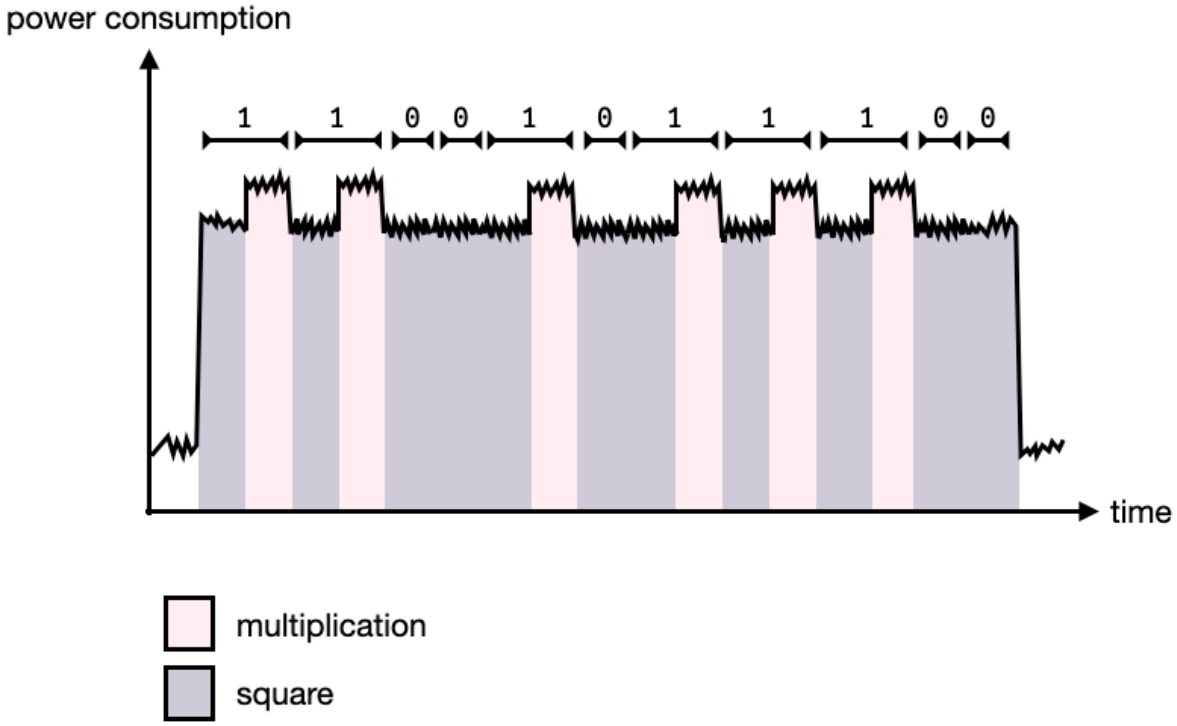


Figure 13.6 Some cryptographic algorithms leak so much information via their power consumption that a simple power analysis of a single power trace (a measure of the power consumed in time) can leak the private key of the algorithm. For example, this figure represents a trace of an RSA exponentiation (the message being exponentiated to the private exponent, see chapter 6) done with the square-and-multiply algorithm which looks at the bits of the private exponent one by one and works by applying a square and multiply operation if a bit is 1, and a square operation if a bit is 0. The multiplication is obviously consuming more power, hence the clarity of the power trace.

Power is not the only physical side-channel, some attacks rely on electromagnetic radiations, vibrations, and even the sound emitted by the hardware.

Let me still mention two other non-physical side-channels. I know we are in a hardware-focused chapter, but these non-physical side-channels attacks are important as they need to be mitigated in many real-world cryptographic applications:

Errors. Errors returned can sometimes leak critical information. For example, in 2018 the ROBOT attack figured out a way to exploit the Bleichenbacher attack (mentioned in chapter 6) on a number of servers that implemented RSA PKCS#1 v1.5 decryption in the TLS protocol (covered in chapter 9).[50] Bleichenbacher's attack only works if you can distinguish when an RSA encrypted message sent has a valid padding or not. To protect against that attack, safe implementations will perform the padding validation in constant-time and avoid returning early if it detects that the padding is invalid early on. Yet, if at the very end of the padding validation the implementation returns a different error to the client based on the validity of the padding, then this was all for nothing.

Memory Access. Accessing memory can take more or less time depending if the data was

previously accessed or not, this is due to the numerous layers of caching that exist in a computer. For example if the CPU needs something it will first check if it has been cached in its internal memory, if not it will then reach into caches that are further and further away from it. The further away the cache, the more time it'll take. Not only that, but some caches are specific to a core (L1 cache, for example) while some caches are shared amongst cores in a multicore machine (L3 cache, RAM, disk). **Cache attacks** exploit the fact that it is possible for a malicious program to run on the same machine and to use the same cryptographic library as a sensitive cryptographic program. For example, many cloud services host different clients' virtual servers on the same machine, and many servers use the OpenSSL library for cryptographic operations or for serving TLS pages. These malicious programs find ways to evict parts of the library that have been loaded in a cache shared with the victim's process, and then periodically measure the time it takes to reread some parts of that library. If it took a long time, then the victim did not execute this part of the program, if it didn't take a long time then the victim did access this part of the program and repopulated the cache (to avoid having to refetch this to a far away cache, or worse from disk). What you obtain is a trace that resembles a power trace! And it is indeed exploitable in similar ways.

OK that's enough for side-channel attacks. If you're interested in attacking cryptography via these side-channels there are better resources than this book. In this section, I will talk only about software mitigations that cryptographic implementations can and should implement to protect against side-channel attacks in general. This whole field of study is called **leakage-resilient cryptography** as the cryptographer's goal here is to not leak anything.

Note that defending against physical attackers is an endless battle, which explains why many of these mitigations are proprietary and akin to obfuscation. This section is obviously not exhaustive, but should give you an idea of the type of things applied cryptographers are working on to address side-channel attacks.

13.4.1 Constant-Time Programming

The first line of defense for any cryptographic implementation is to implement its cryptographic sensitive parts (think any computation that involves a secret) in constant-time. It is obvious that implementing something in constant-time cancels timing attacks, but this also gets rid of many classes of attacks like cache attacks and simple power analysis attacks.

How to implement something in constant-time? Never **branch**. In other words: no matter what the input is, always do the same thing.

For example, this is how the Golang language implements a **constant-time comparison of authentication tags for the HMAC algorithm**. Intuitively, if two bytes are equal, then their

XOR will be 0. If this property is verified for every pair of bytes we are comparing, then ORing them will also lead to a 0 value (and a non-zero value otherwise). Warning: it can be quite disconcerting to read this code if this is the first time you're looking at constant-time tricks :)

Listing 13.1 How Golang implements a constant-time comparison between two bytearrays.

```
func ConstantTimeCompare(x, y []byte) byte {
    if len(x) != len(y) {   ❶
        return 0   ❶
    }   ❶

    var v byte   ❷
    for i := 0; i < len(x); i++ {   ❷
        v |= x[i] ^ y[i]   ❷
    }   ❷

    return v   ❸
}
```

❶ There is no point comparing two strings in constant-time if they are of different lengths.

❷ Here is where the magic happens, the loop OR accumulates the XOR of every byte into a value v.

❸ Returns 0 only if `v` is equal to 0, and returns a non-zero value otherwise.

For a MAC authentication tag comparison, it is enough to stop here to check if the result is 0 or not by branching (using a conditional expression like `if`).[51]

Another interesting example are **scalar multiplications** in elliptic curve cryptography, which as you've learned in chapter 5 consists of adding a point to itself `x` number of times (where `x` is what we call a scalar). This process can be somewhat slow, and thus clever algorithms exist to speed this part up. One of the popular one is called **Montgomery's ladder** and is pretty much the equivalent to the RSA's square-and-multiply algorithm I've mentioned earlier (but in a different group). Montgomery ladder's algorithm alternates between addition of two points and doubling of a point (adding the point to itself). Both algorithms have a simple way to mitigate timing attack: by not branching and always doing both operations. (And this is why the RSA exponentiation algorithm in constant-time is usually referred to as **square and multiply always**.)

> **NOTE** I have mentioned that signature schemes can go wrong in multiple ways in chapter 7, and that key recovery attacks exist against implementations that leak a few bytes of the nonces they use in signature schemes like ECDSA. This is what happened for real in the Minerva [52] and TPM Fail [53] attacks that happened around the same time. Both attacks found out that a number of devices were vulnerable to these kinds of attacks, due to amount of timing variation the signing operation would take.

In practice timing attacks are not always straightforward to mitigate, as it is not always clear if

CPU instructions for multiplications or conditional moves are always constant time, and it is not always clear how the compiler will compile high-level code when used with different compilation flags. For this reason, a manual review of the assembly generated is sometimes performed in order to obtain more confidence in the constant-time code written. Different tools also exist (with different degrees of result) to analyze constant-time code.[54]

13.4.2 Don't use the secret! Masking and blinding

Another common way of thwarting, or at least confusing attackers, is to add layers of indirection to any operations involving secrets. One of these techniques is called **blinding**, and is often possible thanks to the arithmetic structure of public-key cryptography algorithms.

You've seen blinding being used in oblivious algorithms like password-authenticated key exchange (PAKE) algorithms in chapter 11, and we can use blinding in the same way where we want the oblivious party to be the attacker observing leaks from our computations.

Let's talk about RSA as an example.

Remember, RSA decrypts by taking a ciphertext `c` and raising it to the private exponent `d`, where the private exponent `d` cancels the public exponent `e` which was used to compute the ciphertext as `m^e mod N`. If you don't remember the details, make sure to consult chapter 6.

One way to add indirection is to perform the decryption operation on a value that is not the ciphertext known to the attacker. This method is called **base blinding** and goes like this:

1. Generate a random blinding factor `r`.
2. Compute $message' = (ciphertext \times r^e)^d \bmod N$.
3. Finally, unblind the result by computing $message = m' \times r^{-1} \bmod N$ where r^{-1} is the inverse of `r`.

This method blinds the value being used with the secret, but we can also blind the secret itself. For example, elliptic curve scalar multiplication is usually used with a secret scalar. But as computations take place in a cyclic group, adding a multiple of order to that secret does not change the computation result. This technique is called **scalar blinding** and goes like this:

1. Generate a random value `k`.
2. Compute a scalar $d' = d + k \times order$ where `d` is the original secret scalar and `order` is its order.
3. To compute `Q = [d]P`, instead compute `Q = [d'] P` which will result in the same point.

Usually all of these techniques have been proven to be more or less efficient, and are often used in combinations with other software and hardware mitigations.

In symmetric cryptography, another somewhat similar technique called **masking** is used.

The concept of masking is to transform the input (the plaintext or ciphertext in the case of a cipher) before passing it to the algorithm. For example, by XORing it with a random value. The output is then unmasked in order to obtain the final correct output.

As any intermediate state is thus masked, this provides the cryptographic computation some amount of decorrelation from the input data, and makes side-channel attacks much more difficult.

Note that the algorithm has to be aware of this masking to correctly perform internal operations while keeping the correct behavior of the original algorithm.

13.4.3 What about fault attacks?

I've previously talked about **fault attacks**, a more intrusive type of side-channel attacks that modify the execution of the algorithm by inducing faults.

Injecting faults can be done in many creative ways physically, by increasing the heat of the system for example, or even by shooting lasers at calculated points in the targeted chip.

Surprisingly, faults can also be induced via software. An example was found independently in the Plundervolt and V0LTpwn attacks, which managed to change the voltage of a CPU to introduce natural faults. This also happened in the infamous rowhammer attack, which found out that repeatedly accessing memory of some DRAM devices could flip nearby bits.

These types of attacks can be difficult to achieve, but are extremely powerful. In cryptography, computing a bad result can sometimes leak the key. This is, for example, the case with RSA signatures that are implemented with some specific optimizations.

While it is impossible to fully mitigate these attacks, there exist some techniques that can increase the complexity of a successful attack. For example, by computing the same operation several times and comparing the results to make sure they match before releasing it, or by verifying the result before releasing it (for signatures, one can verify the signature via the public key before returning it).

Fault attacks can also have dramatic consequences against random number generators. One easy solution there is to use algorithms that do not use new randomness every time they run. For example, in chapter 7 you learned about EdDSA, a signature algorithm that requires no new randomness to sign as opposed to the ECDSA signature algorithm.

13.5 Summary

- The threat today is not just an attacker intercepting messages over the wire, but an attacker stealing or tampering with the device that runs your cryptography. So-called Internet of Things (IoT) devices often run into this type of threats and are by default unprotected against sophisticated attackers. More recently cloud services are also considered in the threat model of their users.
- **Hardware can help protect cryptography applications and their secrets in a highly-adversarial environment**. One of the ideas is to provide a device with a tamper-resistant chip to store and perform crypto operations. That is, if the device falls in the hands of an attacker, extracting keys or modifying the behavior of the chip will be hard.
- It is generally accepted that one has to combine different software and hardware techniques to harden cryptography in adversarial environments. But hardware-protected cryptography is not a panacea, it is merely **defense-in-depth**, effectively slowing down and **increasing the cost of an attack**. Adversaries with unlimited time and money will always break your hardware.
- Decreasing the impact of an attack can also help detter attackers. This must be done by designing a system well, for example by making sure that the compromise of one device does not imply a compromise of all devices.
- While there are many hardware solutions, the most popular ones are:
    - **Smart cards** were one of the first such secure microcontrollers that could be used as a micro computer to store secrets and perform cryptographic operations with them. These are supposed to use a number of techniques to discourage physical attackers.
    - The concept of a smart card was generalized as a **secure element**, which is a term employed differently in different domains, but boils down to a smart card that can be used as a coprocessor in a greater system that already has a main processor.
    - Trusted Platform Modules (TPMs) are chips that follow the TPM standard, more specifically they are a type of secure element with a specified interface. A TPM is usually a secure chip directly linked to the motherboard and perhaps implemented using a secure element. While it does not allow running arbitrary programs like some secure elements, smart cards, and HSMs do, it enables a number of interesting applications for devices as well as user applications.
    - Hardware security tokens are user-oriented devices that can be useful to store keys and enforce user intent at the same time. They are used for multi-factor authentication, cryptocurrency wallets, and other use cases, and are often built with a secure element.
    - Hardware Security Module (HSM) is a term often used to refer to external, bigger and faster secure elements. They do not follow any standard interface, but usually implement the PKCS#11 standard for cryptographic operations. HSMs can be certified with different levels of security via some NIST standard (FIPS 140-2 and soon 140-3).
    - Trusted Execution Environment (TEE) is a way to segregate an execution environment between a secure one and an insecure one. This is usually implemented via special hardware, and is found integrated as part of many modern CPUs (extending their instruction set).
- Hardware is not enough to protect cryptographic operations in highly-adversarial environments, as software and hardware side-channel attacks can still exploit leakage that

occurs in different ways (timing, power consumption, electromagnetic radiations, and so on). In order to defend against these side-channel attacks cryptographic algorithms implement software mitigations:

  - Serious cryptographic implementations are based on constant-time algorithms and avoid all branching, as well as memory accesses that depend on secret data.
  - Mitigation techniques based on blinding and masking will decorelate sensitive operations from either the secret or the data known to be operated on.
  - Fault attacks are harder to protect against. Mitigations include computing an operation several times and comparing, and verifying the result of an operation (for example verifying a signature with the public key), before releasing the result.

- Hardening cryptography in adversarial settings is a never-ending battle. One should use a combination of software and hardware mitigations to increase the cost and the time for a successful attack up to a desired accepted risk. One should also decrease the impact of an attack by using unique keys per device, and potentially unique keys per cryptographic operation.

14 Post-quantum cryptography

This chapter covers

- Quantum computers and how they impact today's cryptographic algorithms.
- Post-quantum cryptography and this new field's attempt at providing algorithms that can resist quantum computers.
- The post-quantum algorithms that you can use today, and the ones that you might use in the future.

In the middle of 2015, the National Security Agency (NSA) took everybody by surprise after announcing their plans to transition to quantum resistant algorithms in their Commercial National Security Algorithm Suite (CNSA Suite, previously known as "Suite B") of approved cryptographic algorithms (a requirement for protecting government-related documents and applications).

> *For those partners and vendors that have not yet made the transition to Suite B elliptic curve algorithms, we recommend not making a significant expenditure to do so at this point but instead to prepare for the upcoming quantum resistant algorithm transition. [...] Unfortunately, the growth of elliptic curve use has bumped up against the fact of continued progress in the research on quantum computing, which has made it clear that elliptic curve cryptography is not the long term solution many once hoped it would be. Thus, we have been obligated to update our strategy.*
>
> *– National Security Agency Cryptography Today (2015)*

Here, the NSA is referring to a new way of building computers (called **quantum computers**) which could affect today's cryptography algorithms, and a new way of building cryptographic algorithms that could withstand attacks from such computers (called **quantum-resistant or post-quantum cryptography**). While the idea of quantum computing is not new: building a computer based on physical phenomena studied in the field of quantum mechanics, the idea has

witnessed a huge boost in research grants as well as experimental breakthroughs in recent years. Of course, one needs to remember that the government's need for confidentiality most often exceeds the needs of individuals and private companies. It is not crazy to think that the government might want to keep some top secret data classified for more than 50 years. Nevertheless, this has puzzled many cryptographers,[55] who have been wondering why we would protect ourselves against something that doesn't exist yet, or might never exist. Does the NSA know something we don't? Are quantum computers really going to break cryptography today? And what exactly are quantum computers and post-quantum cryptography?

In this chapter I will attempt to answer all your questions!

14.1 What are quantum computers and why are they scaring cryptographers?

Since NSA's announcement, quantum computers have repeatedly made the news, as many large companies like IBM, Google, Alibaba, Microsoft, Intel, and so on have invested significant resources into researching them. But what are these quantum computers and why are they so scary?

It all begun with **quantum mechanics** (also called **quantum physics**), a field of physics that studies the behavior of small stuff (think atoms and smaller). As this is the basis of quantum computers, this is where our investigation will start.

> *There was a time when the news-pa-pers said that only twelve men un-der-stood the the-ory of rel-a-tiv-ity. I do not be-lieve there ever was such a time. There might have been a time when only one man did, be-cause he was the only guy who caught on, be-fore he wrote his pa-per. But af-ter peo-ple read the pa-per, a lot of peo-ple un-der-stood the the-ory of rel-a-tiv-ity in some way or other, cer-tainly more than twelve. On the other hand, I think I can safely say that no-body un-der-stands quan-tum me-chan-ics.*
>
> *– Richard Feyn-man The Char-ac-ter of Phys-i-cal Law (1965)*

14.1.1 Quantum mechanics, the study of the small

Physicists have long thought that the whole world is **deterministic**. Deterministic like our cryptographic pseudo-random number generators: if you knew how the universe worked, and if you had a computer large enough to compute the "universe function," all you would need is the seed (the information contained in the Big Bang) and you could predict everything from there. Yes **everything**, even the fact that merely 13.7 billion years after the start of the universe you were going to read this line. In such a world, there is no room for randomness: every little decision that you're taking is predetermined by past events that came even before you were born.

While this view of the world has bemused many philosophers—"do we really have free will, then?" they ask—an interesting field of physics started growing in the 90s which has puzzled

many scientists since then: the field of **quantum physics** (also called **quantum mechanics**). It turns out that very small objects (think atoms and smaller) tend to behave quite differently from what we've observed and theorized so far (using what we call classical physics).

At this (sub-)atomic scale, particles seem to behave like waves sometimes, in the sense that different waves can superpose to merge into a bigger wave, or cancel each other, for a brief moment.

One measurement we can perform on particles like electrons is their **spin**; for example, we can measure whether an electron is spinning "up" or "down." So far, nothing too weird. What's weird is that quantum mechanics says that a particle can be in these two states at the same time, spinning up **and** down. We say that the particule is in **quantum superposition**. This special state can be induced manually using different techniques depending on the type of the particule. A particule can remain in a state of superposition, that is, until we measure it; in which case the particle **collapses** into only one of these possible states (spinning up or down). This quantum superposition is what quantum computers end up using: instead of having a bit that can either be a 1 or a 0, a **quantum bit** or **qubit** can be both 0 and 1 at the same time.

Even weirder, quantum theory says that it is only when a measurement happens, and not before, that a particle in superposition decides at random which state it is going to take (each state having a 50% chance of being observed). If this seems weird, you are not alone; many physicists could not conceive how this would work in the deterministic world they had painted. Einstein, convinced that something was wrong with this new theory, once said "God does not play dice."

Yet cryptographers were interested, as this was a way to finally obtain **truly** random numbers! This is what **quantum random number generators (QRNGs)** do, by continuously setting particles like photons in a superposed state, and then measuring them.

Physicists have also theorized what quantum mechanics would look like with objects at our scale, with the famous thought experiment of **Schrödinger's cat** where a cat in a box is both dead and alive, that is until an "observer" takes a look inside (which has led to many debates on what exactly constitutes an observer).

A cat is penned up in a steel chamber, along with the following device (which must be secured against direct interference by the cat): in a Geiger counter, there is a tiny bit of radioactive substance, so small, that perhaps in the course of the hour one of the atoms decays, but also, with equal probability, perhaps none; if it happens, the counter tube discharges and through a relay releases a hammer that shatters a small flask of hydrocyanic acid. If one has left this entire system to itself for an hour, one would say that the cat still lives if meanwhile no atom has decayed. The first atomic decay would have poisoned it. The psi-function of the entire system would express this by having in it the living and dead cat (pardon the expression) mixed or smeared out in equal parts.

– Erwin Schrödinger The present situation in quantum mechanics (1935)

All of that is of course highly unintuitive to us, due to the fact that we never encounter quantum behavior in our day-to-day lives.

Now, let's add even more weirdness! Sometimes particles interact with each other (for example, by colliding into one another) and end up in a state of strong correlation, where it is impossible to describe one particle without the others. This phenomenon is called **quantum entanglement**, and it is the secret sauce behind the performance boost of quantum computers. If, let's say, two particles are entangled, then when one of them is measured, both particles collapse, and the state of one is known to be perfectly correlated to the state of the other. OK that was confusing. Let's take an example: if two electrons are entangled, and one of them is then measured to be spinning up, we know that the other one is then spinning down (but not before the first one is measured), and any such experiment will always turn out the same.

This is hard to believe of course, but more mind-blowing, it was shown that entanglement works even across very long distances. Einstein, Podolsky, and Rosen famously argued that the description of quantum mechanics was incomplete, most probably missing "hidden variables" that would explain entanglement (as in, once the particles are separated, they know exactly what their measurement will be). Einstein, Podolsky, and Rosen described a thought experiment (the EPR paradox, named after the first letters of their last names) in which two entangled particles are separated by a large distance (think light-years away) and then measured at approximately the same time. According to quantum mechanics, the measurement of one of the particles would instantly affect the other particle, which would be impossible, as no information can travel faster than the speed of light according to the theory of relativity (thus the paradox). This strange thought experiment is what Einstein famously called "spooky action at a distance."

John Bell later stated an inequality of probabilities known as Bell's Theorem; the theorem, if shown to be true, would prove the existence of the hidden variables mentioned by the author of the EPR paradox. The inequality was later violated experimentally many many times, enough to convince us that entanglement is real and rejecting the presence of hidden variables. Today, we say that a measurement of entangled particles leads to the particles "coordinating" with each

other, which bypasses the relativistic prediction that communication cannot go faster than light (and indeed, try to think of a way you could use entanglement to devise a communication channel, and you'll see that it is not possible).

For cryptographers, though, the spooky action at a distance meant that we could develop novel ways to perform key exchanges—the idea is called **quantum key distribution (QKD)** Imagine distributing two entangled particles to two peers, who would then measure their respective particles in order to start forming the same key (as measuring one particle would give you information about the measurement of the other). QKD's concept is made even more sexy by the no-cloning theorem, which states that you can't passively observe such an exchange and create an exact copy of one of the particles being sent on that channel. Yet, these protocols are vulnerable to trivial man-in-the-middle attacks, and are sort of useless without already having a way to authenticate data. This flaw has led some cryptographers like Bruce Schneier to state that "QKD as a product has no future."[56]

This is all I'll say about quantum physics, as this is already too much for a book on cryptography. If you don't believe any of the bizarre things that you just read, you are not alone. In his book "Quan-tum Me-chan-ics for En-gi-neers," Leon van Dom-me-len writes "Physics ended up with quan-tum me-chan-ics not be-cause it seemed the most log-i-cal ex-pla-na-tion, but be-cause count-less ob-ser-va-tions made it un-avoid-able."

14.1.2 From the birth of quantum computers to quantum supremacy

It's in 1980 that the idea of quantum computing was born. It is Paul Benioff who is first to describe what a quantum computer could be: a computer built from the observations made in the last decades of quantum mechanics. Later that same year, Paul Benioff and Richard Feynman argue that it is the only way to simulate and analyze quantum systems, short of the limitations of classical computers.

It is only 18 years later that the a quantum algorithm running on an actual quantum computer is demonstrated, for the first time, by IBM. Fast forward to 2011 when D-Wave, a quantum computer company, announces the first commercially available quantum computer, launching an entire industry forward in a quest to create the first scalable quantum computer. There's a long way to go, and a useful quantum computer is still something that hasn't been achieved. The most recent notable result, at the time of this writing, is Google claiming in 2019 to have reached "quantum supremacy" with a 53-qubit quantum computer. Quantum supremacy means that for the first time ever, a quantum computer achieved something that a classical computer couldn't. In 3 minutes and 20 seconds, it performed some analysis that would have taken a classical computer around 10,000 years to finish. That is, before you get too excited, it outperformed a classical computer at a task that wasn't useful. Yet, it is an incredible milestone, and one can only wonder where this will all lead us.

A quantum computer pretty much uses quantum physics' phenomena like superposition and entanglement, the same way classical computers use electricity, to perform computations. Instead of bits, quantum computers use **quantum bits** or **qubits** which can be transformed via **quantum gates** to set them to specific values, or put them in a state of superposition and even entanglement. This is somewhat similar to how gates are used in circuits in classical computers. Once a computation is done, the qubits can be measured in order to be interpreted in a classical way: as 0s and 1s. At that point the result can then be interpreted further with a classical computer in order to finish a useful computation.

In general, N entangled qubits contain information equivalent to 2^N classical bits. But measuring the qubits at the end of a computation only gives you N number of 0s or 1s. Thus, it is not always clear how a quantum computer can help, and quantum computers are only found to be useful for a limited number of applications. It is possible that they will appear more and more useful as people find out clever ways to leverage their power.

Today, you can already use a quantum computer, and that from the comfort of your home. Services like https://quantum-computing.ibm.com allow you to build quantum circuits and execute them on real quantum computers hosted in the cloud. Of course such services are quite limited at the moment (early 2021), with only a few qubits available. Still, it is quite a mind-blowing experience to create your own circuit and wait for it to run on a real quantum computer; and all of that for free.

14.1.3 The impact of Grover and Shor's algorithms on cryptography

Unfortunately, as I said earlier quantum computers are not useful for every type of computation, and thus are not a more powerful drop-in replacement for our classical computers. But then, what are they good for?

In 1994, at a time where the concept of a quantum computer was just a thought experiment, Peter Shor proposed a quantum algorithm to solve the discrete logarithm problem and the factorization problem. Shor had the insight early that a quantum computer could be used to quickly compute solutions to problems that could be related to hard problems seen in cryptography. It turns out that there exists an efficient quantum algorithm that can help in finding a number *period* such that $f(x + period) = f(x)$ for any given x. For example, finding the value *period* such that $g^{x+period} = g^x \bmod N$. This in turn leads to algorithms that can efficiently solve the factorization and the discrete logarithm problems, effectively impacting algorithms like RSA (covered in chapter 6) and Diffie-Hellman (covered in chapter 5). **Shor's algorithm** is devastating for **asymmetric cryptography**, as most of the asymmetric algorithms in use today rely on the discrete logarithm or the factorization problem. Most of what you've seen throughout this book actually.

You could think that the discrete logarithm problem and the factorization problem are still hard problems in mathematics, and that we could maybe increase the size of our algorithms'

parameters in order to upgrade their defense against quantum computers. Unfortunately, it was shown in 2017 by Bernstein and others that raising parameters would work, but it would be highly impractical. The research estimated that RSA could be made quantum resistant by increasing its parameters to 1 terabyte. Unrealistic, to say the least.

> *Shor's algorithm shatters the foundations for deployed public-key cryptography: RSA and the discrete-logarithm problem in finite fields and elliptic curves. Long-term confidential documents such as patient health-care records and state secrets have to guarantee security for many years, but information encrypted today using RSA or elliptic curves and stored until quantum computers are available will then be as easy to decipher as Enigma-encrypted messages are today.*
>
> *– PQCRYPTO: Initial recommendations of long-term secure post-quantum systems (2015)*

For **symmetric cryptography**, things are much less worrisome. **Grover's algorithm** was proposed in 1996 by Lov Grover as a way to optimize a search in an unordered list. A search in an unordered list of N items takes $N/2$ operations on average with a classical computer; it would take $N^{1/2}$ operations with a quantum computer. Quite a speed up.

Grover's algorithm is quite a versatile tool that can be applied in lots of ways in cryptography; for example, as a cryptanalysis brute-force to find a symmetric key, or a collision in a hash function. To exhaustively search a key of 128 bits, Grover's algorithm would run in 2^{64} operations on a quantum computers, as opposed to 2^{127} on a classical computer. This is quite the scary statement for all of our symmetric cryptography algorithms, but simply bumping parameters from 128-bit to 256-bit is enough to counter Grover's attack. Thus, if you want to protect your symmetric cryptography against quantum computers today, you can simply use SHA-3-512 instead of SHA-3-256, AES-256-GCM instead of AES-128-GCM, and so on.

To summarize: symmetric cryptography is mostly fine, asymmetric cryptography is not. This is not ideal, to say the least, as symmetric cryptography is often preceded by a key exchange or something vulnerable to a quantum computer. So is this the end of cryptography as we know it? Not so fast…

14.1.4 Post-quantum cryptography, the defense against quantum computers

Fortunately, this was not the end of the world for cryptography. The community quickly reacted by organizing itself and by researching old and new algorithms that would not be vulnerable to Shor's and Grover's attacks. The field of **post-quantum cryptography** was born.

Standardization efforts exist in different places on the internet, but the most regarded effort is from the NIST, who in 2016 started a post-quantum cryptography standardization process.

It appears that a transition to post-quantum cryptography will not be simple as there is unlikely to be a simple "drop-in" replacement for our current public-key cryptographic algorithms. A significant effort will be required in order to develop, standardize, and deploy new post-quantum cryptosystems. In addition, this transition needs to take place well before any large-scale quantum computers are built, so that any information that is later compromised by quantum cryptanalysis is no longer sensitive when that compromise occurs. Therefore, it is desirable to plan for this transition early.

– Post-Quantum Cryptography page of the NIST standardization process (2016)

Since the NIST started this process, 82 candidates applied and 3 rounds have passed, narrowing down the list of candidates to 7 finalists and 8 alternate finalists (unlikely to be considered for standardization, but unique enough to be a good option if one of the paradigms used by the finalists end up being broken).

The NIST standardization effort seeks to replace the most common type of asymmetric cryptography primitives, which are:

1. Signature schemes.
2. Asymmetric encryption (which can easily serve as key exchange primitives, as you've seen in chapter 6).

In the rest of this chapter, I will go over the different types of post-quantum cryptography algorithms that are being considered for standardization, and point out which ones you can make use of today.

14.2 Hash-based signatures: don't need anything but a hash function

While all practical signature schemes seem to use hash functions, there exist ways to build signature schemes that only make use of hash functions, and nothing else. Even better, they tend to rely only on the pre-image resistance of hash functions, and not their collision-resistance. This is quite an attractive proposition, as a huge part of applied cryptography is already based on solid and well-understood hash functions. Modern hash functions are also resistant to quantum computers, which make these hash-based signature schemes naturally quantum-resistant. Let's take a look at what these hash-based signatures are and how they work.

14.2.1 One-time signatures (OTS) with Lamport signatures

On October 18, 1979, Leslie Lamport published his concept of **One-Time Signatures (OTS)**. Most signature schemes rely in part on one-way functions, typically hash functions, for their security proofs. The beauty of Lamport's scheme was that his signature only relied on the security of such one-way functions.

Imagine that you want to sign a single bit set to 0 or 1. You first generate your keypair as so:

1. you generate two random numbers *x* and *y*, which will be your private key
2. you then hash *x* and *y* to obtain two digests *h(x)* and *h(y)*, which you can publish as your public key.

To sign the bit 0 publish *x*, or to sign the bit 1 publish *y*. To verify a signature, simply hash it to check that it matches the right part of the public key! As the name of the scheme indicates, you have to be confident about what you sign, because once you sign a bit you can't reuse that keypair to sign another bit. I illustrate this in figure 14.1.

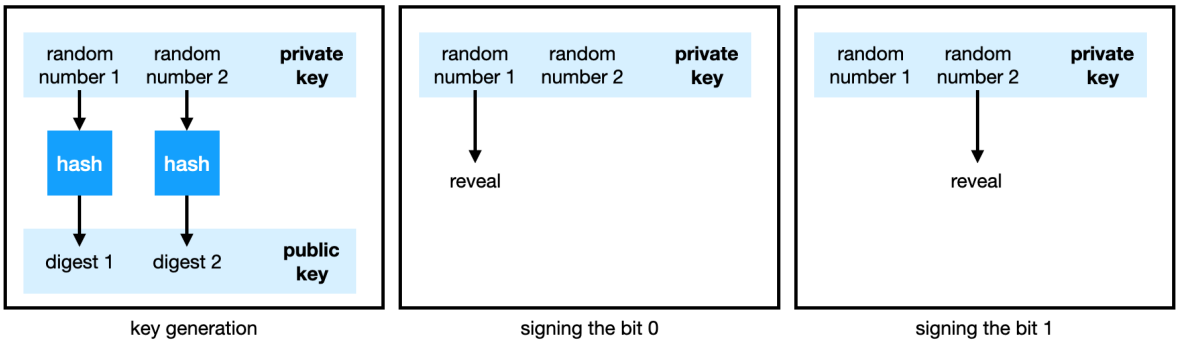


Figure 14.1 A one-time signature (or lamport signature) is a signature scheme based only on hash functions. To generate a keypair that can sign a bit, generate 2 random numbers (which will be your private key) and hash each of them individually to produce the 2 digests of your public key. To sign a bit set to 0, reveal the first random number; to sign a bit set to 1, reveal the second random number.

Signing a bit is not that useful, you say. No problem, a lamport signature works for larger inputs simply by creating more pairs of secrets, one per bit to sign (see figure 14.2). Obviously if your input is larger than 256 bits, you would first hash it and then sign it.

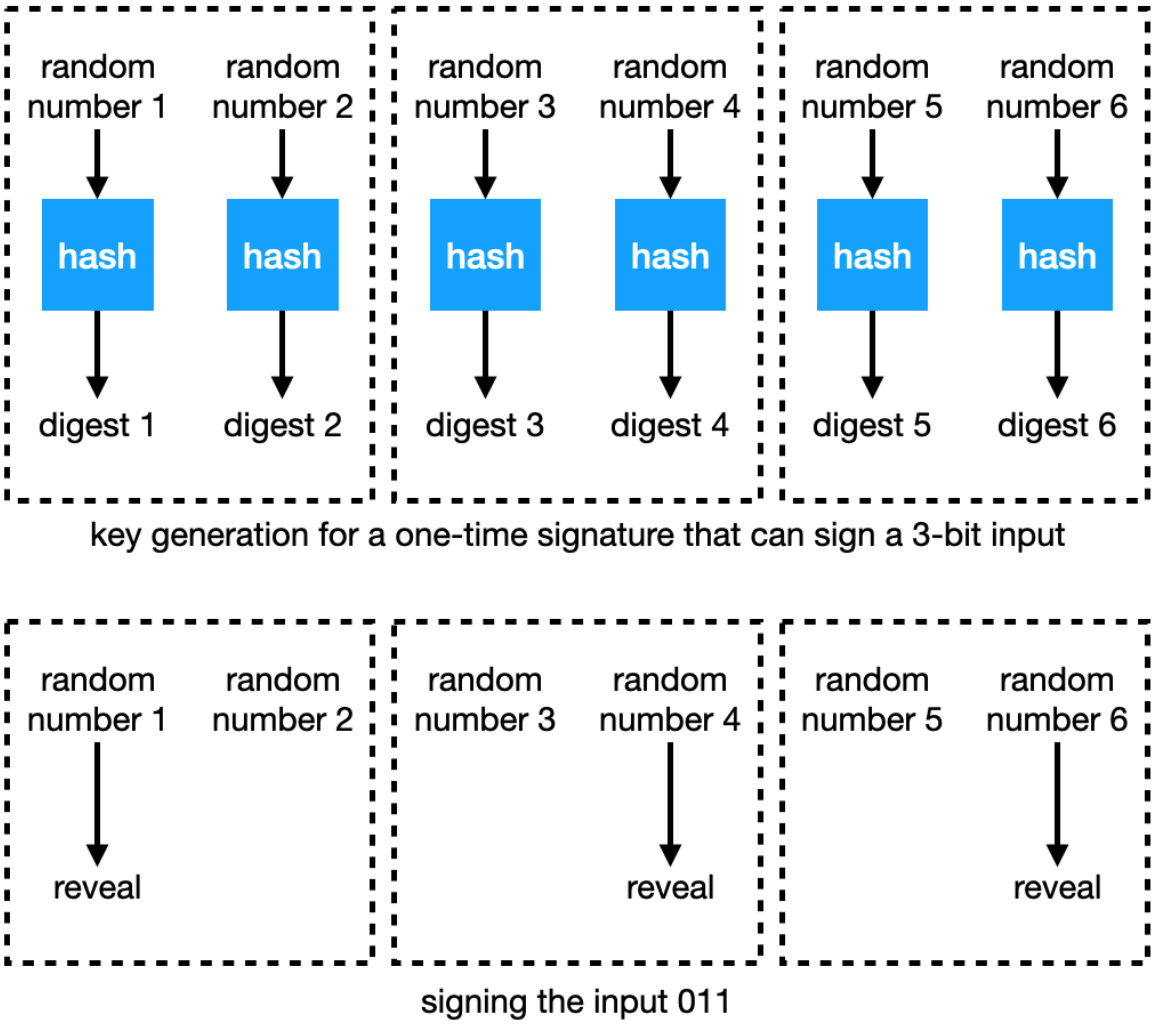


Figure 14.2 To generate a lamport signature keypair that can sign an n-bit message, generate 2n random numbers (which will be your private key) and hash each of them individually to produce the 2n digests of your public key. To sign, go through pairs of secrets and bits of n, revealing the first element to sign a bit set to 0, or the second element to sign a bit set to 1.

A major limitation of this scheme is that you can only use it to sign once. We can improve the situation naively, by generating a large number of one-time keypairs instead of a single one, and making sure to discard a keypair after using it. Not only this makes your public key as big as the number of signatures you think you might end up using, but it also means you have to track what keypairs you've used (or better, get rid of the private keys you've used). In practice, if you know you'll want to sign a maximum of 1,000 messages of 256 bits with digests of 256 bits as well, this would make both your private key and public key $1{,}000 \times (256 \times 2 \times 256)$ bits, which is around 16 megabytes. That's quite a lot for only 1,000 signatures.

Most of the hash-based signature schemes proposed today build on the foundations created by Lamport to allow for many more signatures (sometimes a practically unlimited amount of signatures), stateless private keys (although some proposed schemes are stateful), and more practical parameter sizes.

14.2.2 Smaller keys with Winternitz one-time signatures (WOTS)

A few months after Lamport's publication, Robert Winternitz of the Stanford Mathematics Department proposed to publish hashes of hashes of a secret $h(h(...h(x))) = h^w(x)$ instead of publishing multiple digests of multiple secrets in order to optimize the size of a private key (see figure 14.3). This scheme is called **Winternitz one-time signature (WOTS)** after the author's name.

For example, choosing $w=16$ allows you to sign 16 different values, or in other words inputs of 4 bits. You start by generating a random value x that serves as your private key, and hash it 16 times to obtain your public key $h^{16}(x)$. Now imagine you want to sign the bits *1001* (9 in base 10), just publish the 9th iteration of the hash: $h^9(x)$. I illustrate this in figure 14.3.

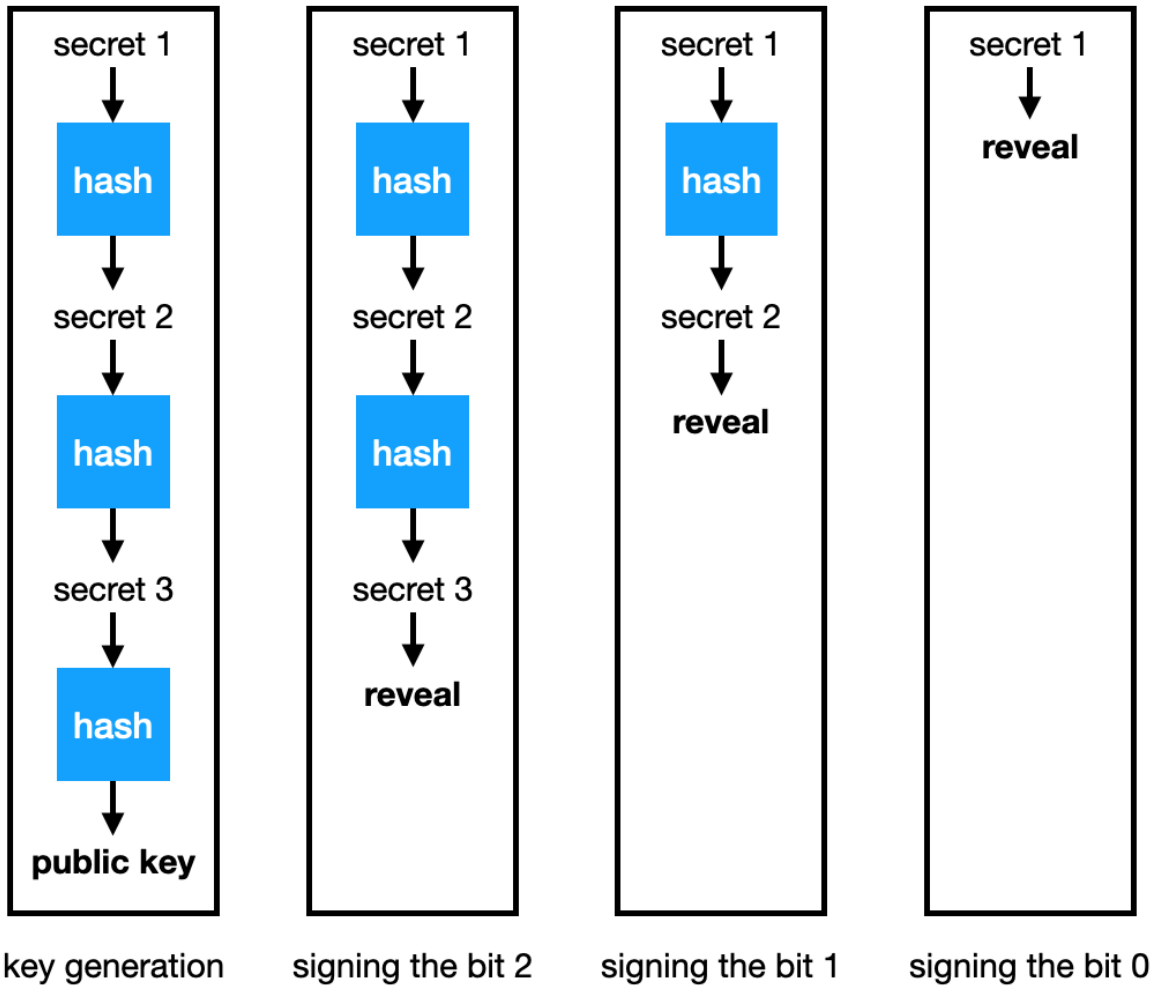


Figure 14.3 The Winternitz one-time signature (WOTS) scheme optimizes lamport signatures by only using one secret that is hashed iteratively in order to obtain many other secrets (and finally a public key). Revealing a different secret allows one to sign a different number.

Take a few minutes to understand how this scheme works. Do you see a problem with it?

One major problem is that this scheme allows for signature forgeries. Imagine that you see someone else's signature for *1001*, which would be $h^9(x)$ according to our previous example. You can simply hash it to retrieve any other iterations like *h^10(x)* or *h^11(x)* which would give you a valid signature for *1010* or *1011*. This can be circumvented by adding a short authentication tag after the message, which you would have to sign as well. I illustrate this in figure 14.4.

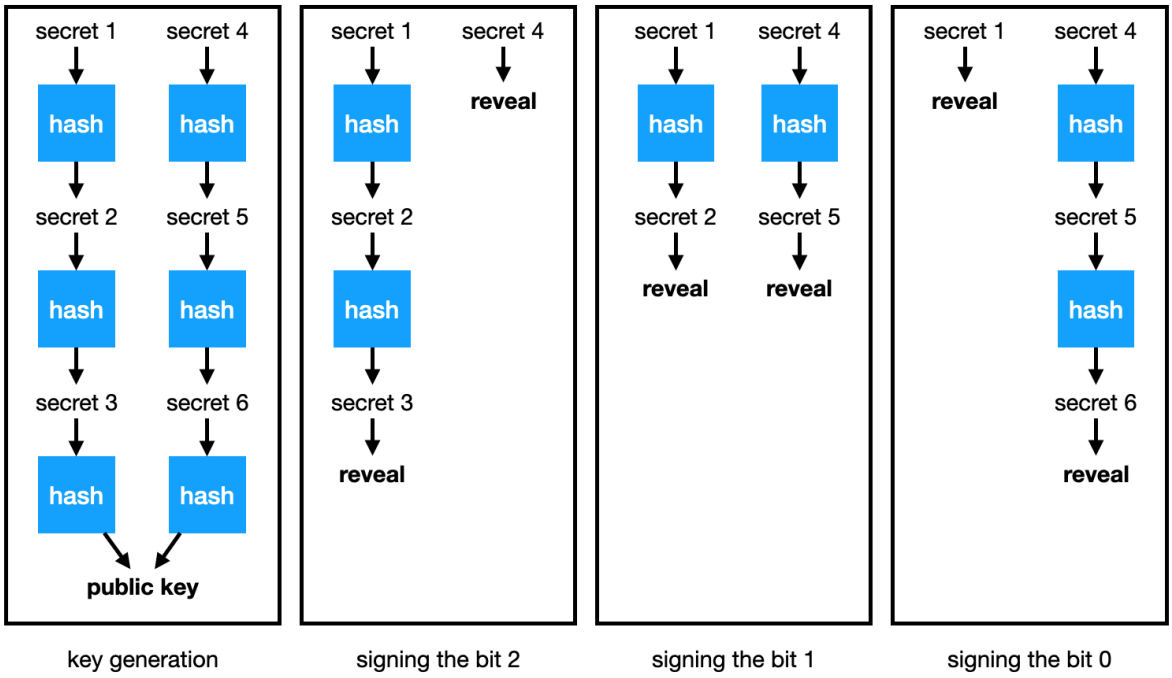


Figure 14.4 WOTS uses an additional signing key to authenticate a signature, in order to prevent tampering. It works like this: when signing, the first private key is used to sign the message, and the second private key is used to sign the complement of the message. It should be clear that in any of the scenarios illustrated, tampering with a signature cannot lead to a new valid signature.

To convince yourself that this solves the forgery issue, try to forge a signature from another signature.

14.2.3 Many-times signatures with XMSS and SPHINCS+

So far, you've seen ways of signing things using only hash functions. While Lamport signatures work, they have large key sizes. WOTS improved on Lamport signatures by reducing the key sizes. Yet, both these schemes still don't scale well as they are both one-time signatures (reuse a keypair and you break the scheme) and thus their parameters linearly increase in size depending on the number of signatures you'll think you need.

> **NOTE** Note that some schemes exist that tolerate reuse of a keypair for a few signatures (instead of a single one). These schemes are called few-time signatures (FTS) and they will break, allowing signature forgeries, if reused too many times. FTS are schemes that relies on low probabilities of reusing the same combination of secrets twice, from a pool of secrets. They are a small improvement on one-time signatures, allowing to decrease the risk of key reuse.

What is one technique you've learned about in this book that compresses many things into one thing? **Merkle trees**! As you may recall from chapter 12, a Merkle tree is a data structure that provides short proofs for questions like: is my data in this Merkle tree? The same Merkle who proposed Merkle trees also invented a signature scheme based on hash functions in the 90s, by compressing a number of one-time signatures into a Merkle tree. The idea is pretty straightforward: each leaf of your tree is the hash of a one-time signature, and the root hash can be used as a public key (reducing its size to the output size of your hash function). To sign, pick a one-time signature that you haven't used previously and use it as explained in section 14.2.2. The signature is the one-time signature, along with the Merkle proof that it belongs in your Merkle tree (all the neighbors). This scheme is obviously stateful, as one should be careful not to reuse one of the one-time signatures in the tree. I illustrate it in figure 14.5.

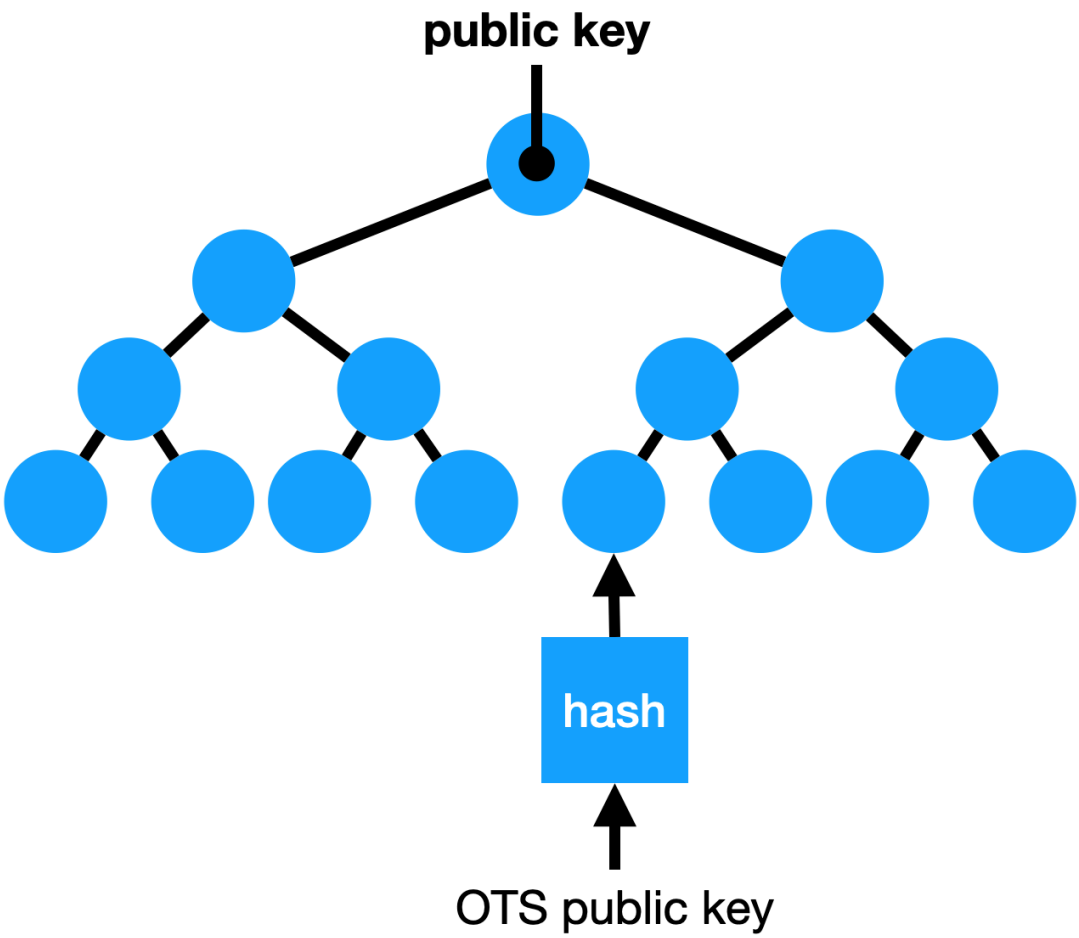


Figure 14.5 Merkle signatures is a stateful hash-based signature scheme that makes use of a Merkle tree to compress many one-time signatures (OTS) public keys into a smaller public key (the root hash). The larger the tree, the more signatures it can produce. Note that signatures now have the overhead of a membership proof, which is a number of neighbor nodes that allow one to verify that a signature's associated OTS is part of the tree.

The **extended Merkle signature scheme (XMSS)**, standardized in RFC 8391, sought to productionize Merkle signatures by adding a number of optimizations to Merkle's scheme. For example, to produce a keypair capable of signing N messages you must generate N OTS private keys. While the public key is now just a root hash, you still have to store N OTS private keys. XMSS reduces the size of the private you hold by deterministically generating each OTS in the tree based on two things: a seed and the leaf position in the tree. This way, you only need to store this seed as private key (instead of all the OTS private keys), and can quickly regenerate any OTS keypair from their position in the tree and the seed. To keep track of which leaf/OTS was used last, the private key also contains a counter which is incremented every time it is used to sign.

Having said that, there's only so much OTS you can hold in a Merkle tree. The larger the tree, the longer it'll take to regenerate the tree in order to sign messages (as you need to regenerate all the leaves to produce a Merkle proof). One way to reduce the amount of OTS private keys that

must be regenerated is to have a smaller tree, but this would obviously defeat the purpose as we are now back to having a limited amount of OTS. The trick is to use a smaller tree, where the OTS in its leaves are not used to sign messages, but used to sign the root hash of other Merkle trees of OTS. This transforms our initial tree into an hypertree—trees of trees—and is one of the variants of XMSS called **XMSS^{MT}**. With XMSS^{MT}, only the trees involved in the path of an OTS needs to be regenerated (based on the same technique). I illustrate this in figure <<hypertree>.

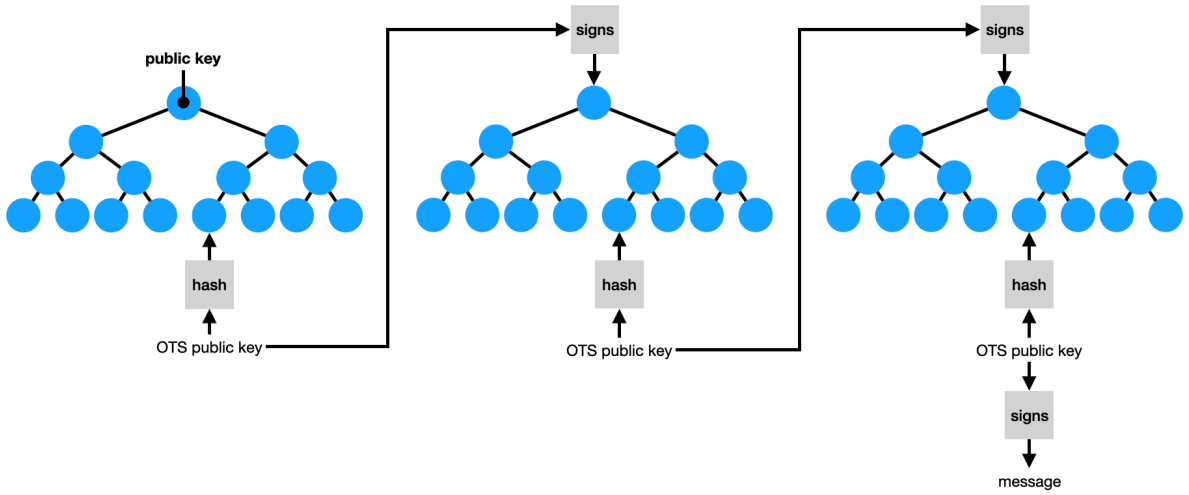


Figure 14.6 The XMSS^{MT} stateful hash-based signature scheme uses multiple trees to increase the amount of signatures supported by the scheme while reducing the work at key generation and signing time. Each tree is deterministically generated only when they are used in the path to the final leaf, containing the OTS that will be used to sign a message.

Note that the statefullness of XMSS and XMSS^{MT} might not be an issue in some situations, but it is not a desirable property in general. Having to keep track of a counter is counterintuitive, as it is not expected from users of mainstream signature schemes. This change of practice can lead to OTS reuse (and thus to signature forgery) in case of misuse. For example, rollbacks to a previous state of the filesystem, or using the same signing key on multiple servers, might induce the same path in the hypertree being used twice to sign a message.

To fix the biggest downside of XMSS—it's statefullness—and expose an interface similar to the signature schemes we're used to, the **SPHINCS+** signature scheme was proposed as part of the NIST's post-quantum cryptography competition. The stateless signature scheme augments XMSS^{MT} with three major changes:

1. **Signing the same message twice leads to the same signature**. In a similar fashion to EdDSA (covered in chapter 7), the path used in the hypertree is deterministically derived, based on the private key and the message. This ensures that signing the same message twice leads to the same OTS (and thus same signature); and since the private key is used, attackers are also unable to predict which path you'll take to sign their messages (if you somehow sign other people's messages).
2. **More trees**. XMSS^{MT} avoids reusing the same OTS twice by keeping track of which OTS was used last. As the whole point of SPHINCS+ is to avoid keeping track of a state,

it needs to avoid collisions when it chooses a path pseudo-randomly. To do this, SPHINCS+ simply uses a much larger amount of OTS, reducing the probability of reusing the same one twice. Since SPHINCS+ also uses an hypertree, this translates into more trees, or in other words a deeper hypertree.

3. **Few-time signatures**. As the security of the scheme is based on the probability of reusing the same path twice, SPHINCS+ also replaces the final OTS used to sign messages with the few-time signatures I mentioned earlier. This way, reusing the same path to sign two different messages still doesn't lead to a break of the signature scheme.

While SPHINCS+ is being considered for standardization in the NIST post-quantum cryptography competition, due to its well-understood security, it's not the main contender. SPHINCS+ is not only slow, its signatures are large in size compared to the proposed alternatives (like lattice-based ones, which we'll learn about later in this chapter). Stateful hash-based signature schemes like XMSS offer much better speed and signature sizes: under 3KB compared to the minimum of 8KB for SPHINCS+. (In terms of public key sizes, both schemes provide sizes similar to pre-quantum signatures schemes like ECDSA and Ed25519.)

Due to the more realistic parameter sizes, and the well-understood security, XMSS is recommended as an early standard by the NIST in **SP 800-208: Recommendation for Stateful Hash-Based Signature Schemes**.

Next, let's take a look at two other ways to build quantum resistant cryptographic primitives. A gental warning: they are much more math-heavy.

14.3 Achieving shorter communication with lattice-based cryptography

A large number of post-quantum cryptography schemes are based on lattices (which I'll explain in a minute). That is, the three types of cryptographic primitives that the NIST is looking into have submissions based on lattices:

- digital signatures: CRYSTALS-DILITHIUM, FALCON
- key-establishment and public-key encryption: CRYSTALS-KYBER, NTRU, SABER, FrodoKEM, NTRUPrime

That is, half of the schemes are based on lattices! This makes lattices the biggest contender for the paradigm that'll get accepted and standardized.

14.3.1 What's a lattice?

First, "lattice-based" probably doesn't mean what you think it means. Take RSA (covered in chapter 6), which we say is based on the factorization problem. This does not mean that we use factorization in RSA, it means instead that factorization is how you attack RSA, and since factorization is hard we say that RSA is secure. It's the same with lattice-based cryptosystems: lattices are structures that have hard problems, and these cryptosystems are safe as long as these problems remain hard. With that being said, what is a lattice? Well, it's like a **vector space** but with integers. If you don't remember what a vector space is, it's the set of all vectors that can be created using:

- **A basis**: a set of vectors (for example, *(0,1)* and *(1,0)*).
- **An operation between vectors**: these vectors can be added together (for example, *(0,1) + (1,0) = (1, 1)*).
- **A scalar operation**: a vector can be multiplied by what we call "scalars" (for example, *3 × (1,2) = (3, 6)*).

So in our example, the vector space contains all the vectors that can be expressed as a **linear combination** of the basis, which translates to any vector that can be written as $a \times (0,1) + b \times (1,0)$ for any a and b. For example, $0.5 \times (0,1) + 3.87 \times (1,0) = (3.87,0.5)$ is in our vector space, so is $99 \times (0,1) + 0 \times (1,0) = (0,99)$, and so on.

A lattice is a vector space where all of the numbers involved are integers. Yup, in cryptography we like integers. I illustrate this in figure 14.7.

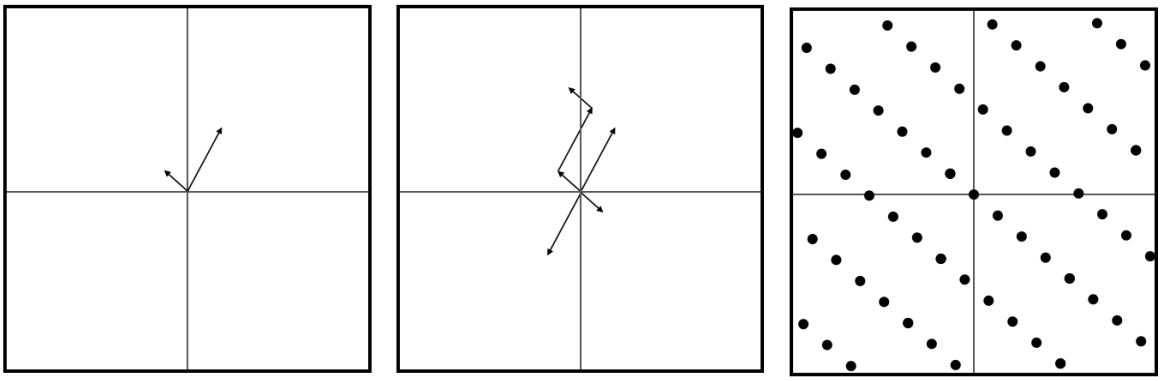


Figure 14.7 On the left, a basis of two vectors are drawn on a graph. A lattice can be formed by taking all of the possible integer linear combinations of these two vectors (middle figure). The resulting lattice can be interpreted as a pattern of points repeating forever in space (right figure).

There are several well-known hard problems in the lattice space, and for each of these problems we have algorithms to solve them. These algorithms are often the best we could think of, but it

doesn't necessarily means that they are efficient or even practical, and thus the problems are said to be hard (at least until an efficient solution is found). The two most well-known hard problems are:

- The **shortest vector problem (SVP)**, which answers the question: what is the shortest non-zero vector in your lattice?
- The **closest vector problem (CVP)**, which given a coordinate that is not on the lattice, finds the closest point to that coordinate on the lattice.

I illustrate both of these problems in figure 14.8.

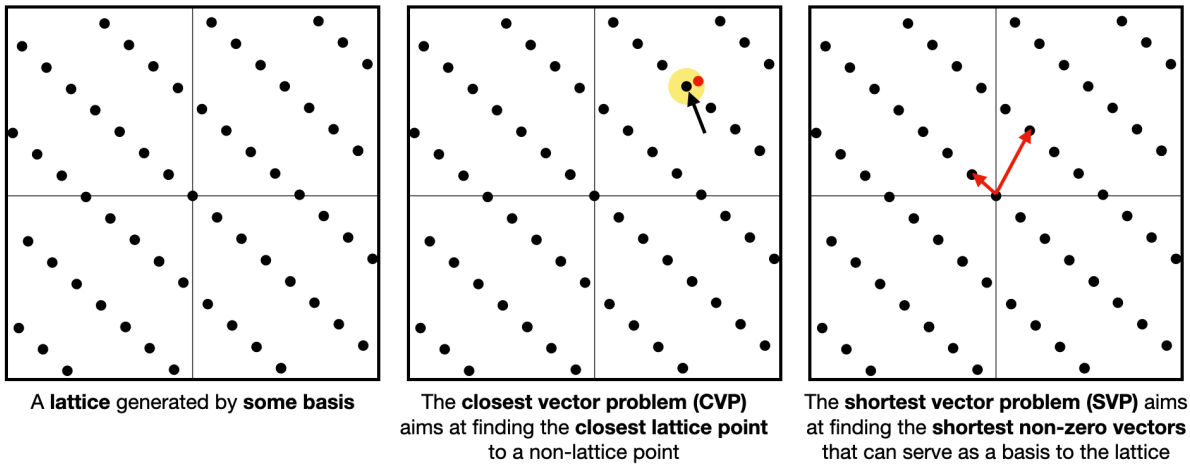


Figure 14.8 An illustration of the two major lattice problems used in cryptography: the shortest vector problem (SVP) and the closest vector problem (CVP).

Generally, we use algorithms like LLL or BKZ to solve both of these problems (CVP can be reduced to SVP), which are algorithms that reduce the basis of a lattice, meaning that they attempt to find a set of vectors that are shorter than the ones given, and that managed to produce the exact same lattice.

14.3.2 Learning with errors (LWE), a basis for cryptography?

In 2005, Oded Regev introduced the **learning with errors (LWE)** problem, which became the basis for many cryptographic schemes (including some of the algorithms in this chapter). Before going further, let's see what thee LWE problem is about.

Let's start with the following equations, which are linear combinations of the same numbers s_0 and s_1:

- $5\ s_0 + 2\ s_1 = 27$
- $2\ s_0 + 0\ s_1 = 6$

We know that by using the **Gaussian elimination** algorithm, we can quickly and efficiently

learn what s_0 and s_1 are, as long as we have enough of these equations.

Now what's interesting, is that if we add some noise to these equations, the problem becomes much harder:

- $5\ s_0 + 2\ s_1 = 28$
- $2\ s_0 + 0\ s_1 = 5$

While it probably isn't too hard to figure out the answer given more of these noisy equations, it becomes a hard problem once you increase the size of the numbers involved and the number of s_i .

This is essentially what the LWE problem is, albeit often stated with vectors instead: imagine that you have a secret vector $\boldsymbol{s}$ with coordinates modulo some large number. Given an arbitrary amount of random vector $\boldsymbol{a_i}$ of the same size, and the computations $\boldsymbol{a_i s} + e_i$ where e_i is a random small error, **can you find the value $\boldsymbol{s}$**?

> **NOTE** **For two vectors v and w, the product vw can be calculated using a dot product, which is the sum of the product of each pair of coordinates. Let's see an example: if $v = (v_0, v_1)$ and $w = (w_0, w_1)$, then $vw = v_0 \times w_0 + v_1 \times w_1$.**

For example, if I use the secret $\boldsymbol{s} = (3, 6)$ and I give you the random vectors $\boldsymbol{a_0} = (5, 2)$ and $\boldsymbol{a_1} = (2, 0)$, we obtain the equations we started the example with.

As I said earlier, lattice-based scheme actually don't make any use of lattices; rather, they are proven secure if the shortest vector problem remains hard (for some definition of hard). The reduction can only be seen if we write the previous equations in a matrix form, as shown in figure 14.9.

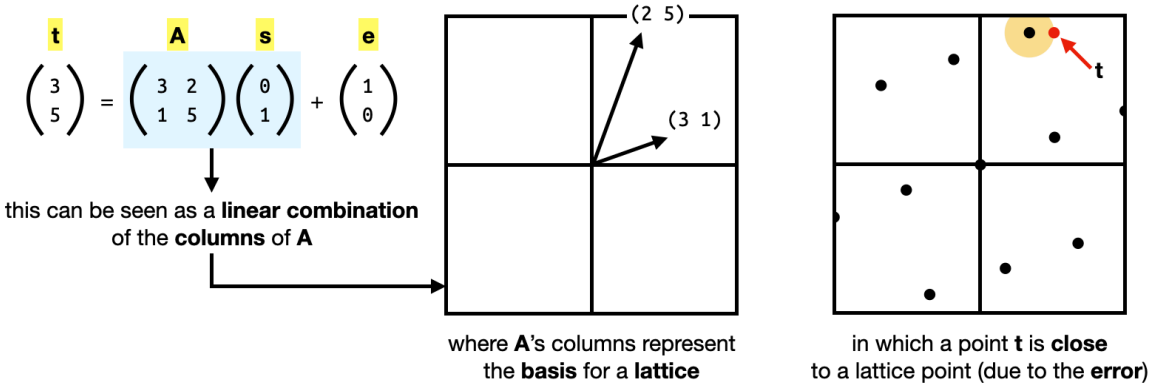


Figure 14.9 The learning with errors problem is said to be a lattice-based construction due the existence of a reduction to a lattice problem: the closest vector problem (CVP). In other words, if we can find a solution to the CVP, then we can find a solution to the LWE problem.

This matrix form is important, as most LWE-based schemes are expressed, and easier to explain, in this form. So take a few minutes to brush up on matrix multiplication. Also, in case you haven't noticed, I've been using some common notation tricks that are quite helpful to read equations that involve matrices and vectors: both are written in bold, and matrices are always uppercase letters. For example, $\boldsymbol{A}$ is a matrix, $\boldsymbol{a}$ is a vector, and a is just a number.

> **NOTE** There exist several variants of the LWE problem (for example, the ring-LWE or module-LWE problems), which are basically the same problem but with coordinates in different types of groups. These variants are often preferred due to the compactness they allow, and the optimizations they enable. Note that the difference between these different variants of LWE does not affect the explanations to follow.

Now that you know what the learning with errors problem is, let's learn about some post-quantum cryptography based on it: the **Cryptographic Suite for Algebraic Lattices (CRYSTALS)**. Conveniently, CRYSTALS encompasses two cryptographic primitives: a key exchange called **Kyber** and a signature scheme called **Dilithium**.

14.3.3 Kyber, a lattice-based key exchange

Two NIST finalist schemes are closely related: **CRYSTALS-Kyber** and **CRYSTALS-Dilithium**, as they are candidates from the same team of researchers, and are both based on the LWE problem. Kyber is a public-key encryption primitive (which can be used as a key exchange primitive), which I will explain in this section. Dilithium is a signature scheme, which I will explain in the next section. Also note that as these algorithms are still in flux, I will only write about the ideas and the intuitions behind both of the schemes.

First, let's assume that all operations happen in a group of integers modulo a large number q. Let's also say that errors and private keys will be sampled (chosen uniformly at random) from a small range centered at 0 that we will call the error range. Specifically, the error range is the range $[-B, B]$ where B is much smaller than q. This is important as some terms will need to be smaller than some value to be considered errors.

Key Generation. To generate the private key, simply generate a random vector $\boldsymbol{s}$ where every coefficient is in the error range. The first part of the public key is a list of random vectors $\boldsymbol{a}_i$ of the same size, and the second part is the associated list of noisy dot products $\boldsymbol{t}_i = \boldsymbol{a}_i\boldsymbol{s} + \boldsymbol{e}_i \; mod \; q$. This is verbatim our LWE problem.

Importantly for the rest, we can rewrite this with matrices: $\boldsymbol{t} = \boldsymbol{A}\boldsymbol{s} + \boldsymbol{e}$ where the matrix $\boldsymbol{A}$ contains the random vectors $\boldsymbol{a}_i$ as rows, and the error vector $\boldsymbol{e}$ contains the individual errors e_i.

Encryption. To perform a key exchange with Kyber, we encrypt a symmetric key of 1 bit (yes a

single bit!) with the scheme; akin to the RSA key encapsulation mechanism seen in chapter 6. This works in four steps:

1. Generate an ephemeral private key vector $\boldsymbol{r}$ (where coefficients are in the error range), and its associated ephemeral public key $\boldsymbol{rA} + \boldsymbol{e_1}$ with some random error vector $\boldsymbol{e_1}$ (using the other peer's $\boldsymbol{A}$ matrix as a public parameter). Note that the matrix multiplication is done on the right which will involve multiplying the vector $\boldsymbol{r}$ with the columns of $\boldsymbol{A}$ (instead of computing $\boldsymbol{Ar}$ which is a multiplication of the vector $\boldsymbol{r}$ with the rows of $\boldsymbol{A}$); it is a detail, but necessary for the decryption step to work.
2. We shift our message to the left by multiplying it with $q/2$, in order to avoid small errors from impacting our message. Note that $q/2$ modulo q usually means q multiplied with the inverse of 2 modulo q, but here it simply means the closest integer to $q/2$.
3. Compute a shared secret with the dot product of our ephemeral private key and the public key of the peer.
4. Encrypt your (shifted) message by adding it to the shared secret as well as a random error e_2. This produces a ciphertext.

We can then send both the ephemeral public key and the ciphertext to the other peer.

Decryption. After receiving both the ephemeral public key and the ciphertext, one can follow these steps to decrypt the message:

1. Obtain the shared secret by computing the dot product of your secret with the ephemeral public key received.
2. Subtract that shared secret from the ciphertext (the result will contain the shifted message and some error).
3. Shift your message back to where it was originally by dividing it with $q/2$, effectively removing the error.
4. Now you can tell that the message is 1 if it is closer to $q/2$ than 0, and 0 otherwise.

Of course, 1 bit is not enough, so current schemes employ different techniques to overcome this limitation. I recapitulate all three algorithms in figure 14.10.

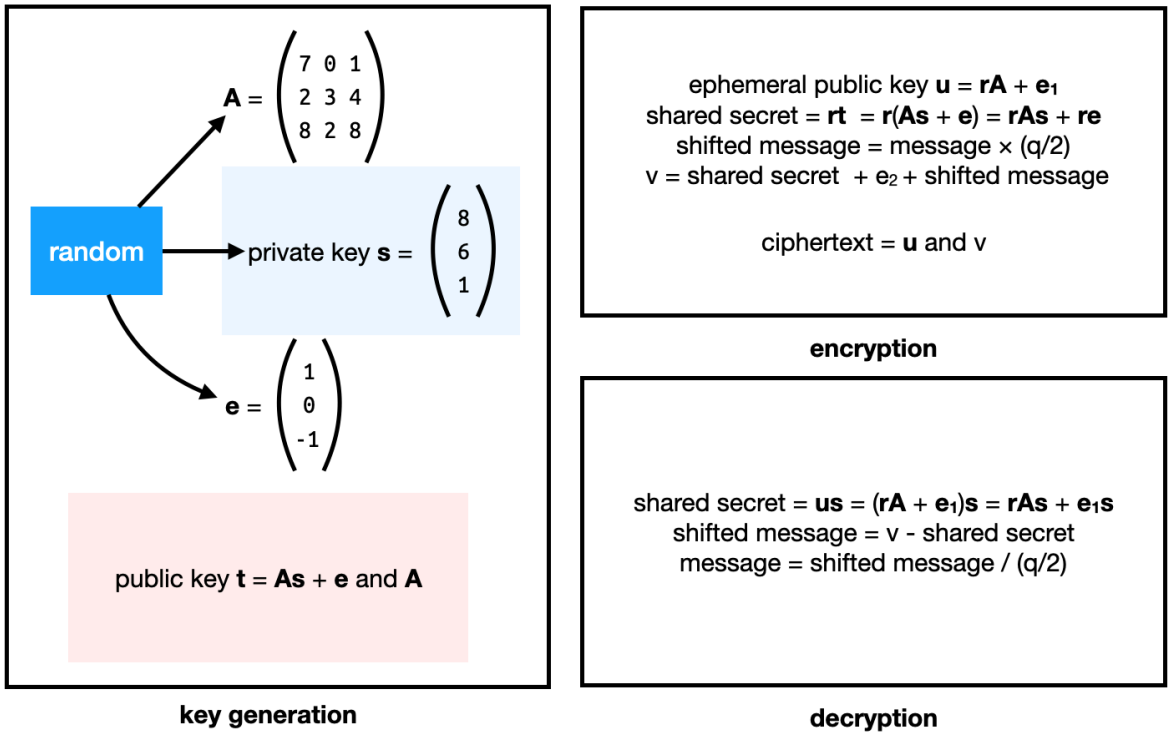


Figure 14.10 The Kyber public-key encryption scheme. Note that the shared secret is approximately the same during encryption and decryption, as re and e_1s are both small values (due to r, s, and the errors being much smaller than q/2). Thus, the last step of the decryption (dividing by q/2, which can be seen as a bitwise shift to the right) gets rid of any discrepancy between the two shared secrets. Note that all operations are done modulo q.

In practice, for a key exchange, the message you encrypt to your peer's public key is a random secret. The result is then derived deterministically from both the secret and the transcript of the key exchange (which includes the peer's public key, your ephemeral key, and the ciphertext).

The recommended parameters for Kyber leads to public keys and ciphertexts of around 1 kylobytes, which is still much bigger than the pre-quantum schemes we use but still in the realm of the practical for most use cases. While time will tell if we can further reduce the communication overhead of these schemes, it seems like so far, post-quantum rhymes with larger sizes.

14.3.4 Dilithium, a lattice-based signature scheme

The next scheme I'll explain, **Dilithium**, is also based on LWE and is the sister candidate of Kyber. As with other types of signatures we've seen before (like Schnorr's signature in chapter 7), Dilithium is based on a zero-knowledge proof that is made non-interactive via the Fiat-Shamir trick.

Key Generation. Dilithium's key generation is close to the scheme we've seen before, except that one keeps the error as part of the private key. One first generates the two random vectors

that will serve as private key s_1 and s_2, then compute the public key as $t = As_1 + s_2$ where A is a matrix obtained in a similar manner as Kyber. The public key is t and A. Note that we consider the error s_2 as being part of the private key, because we need to reuse it every time we sign a message (unlike in Kyber, where the error could be discarded after the key generation step).

To sign, a sigma protocol is created and then converted to a non-interactive zero-knowledge proof via the Fiat-Shamir trick (similar to how the Schnorr identification protocol gets converted to a Schnorr signature in chapter 7). The interactive protocol looks like this:

1. The prover commits on two random vectors y_1 and y_2, by sending $Ay_1 + y_2$.
2. Upon reception of this commit, the verifier responds with a random challenge c.
3. The prover then computes the two vectors $z_1 = cs_1 + y_1$ and $z_2 = cs_2 + y_2$ and sends them to the verifier only if they are small values.
4. The verifier checks if $Az_1 + z_2 - ct$ and $Ay_1 + y_2$ are the same values.

The Fiat-Shamir trick replaces the role of the verifier in step 2, by having the prover generate a challenge from a hash of the message to sign and the committed $Ay_1 + y_2$ value. I recapitulate this transformation in figure 14.11 using a similar diagram I used in chapter 7.

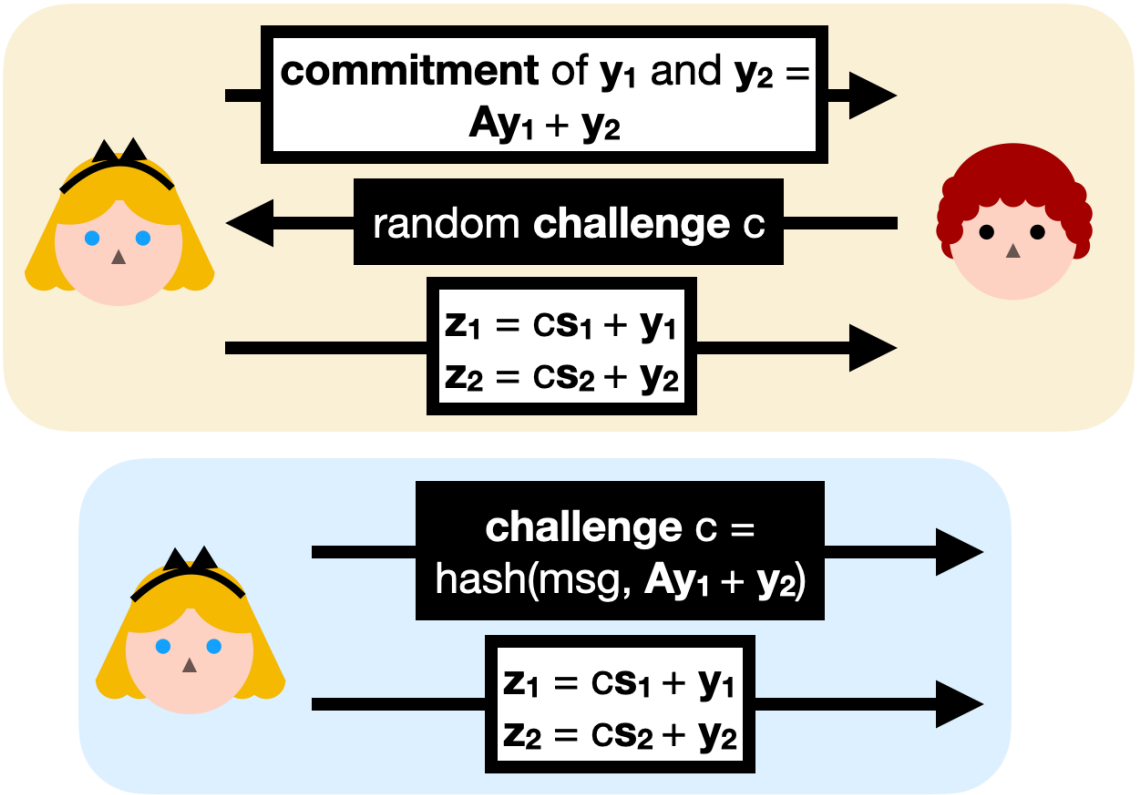


Figure 14.11 A Dilithium signature is a proof of knowledge of a secret vector s made non-interactive via the Fiat-Shamir transformation. The first diagram shows the interactive proof protocol, while the second diagram shows a non-interactive version where the challenge is computed as a commitment of both y and the message to sign.

Again, this is a gross simplification of the signature scheme. Many more optimizations are used in practice to reduce the key and the signature sizes. Usually, these optimizations look at reducing any random data by deterministically generating it from a smaller random value and by reducing non-random data by compressing it (via custom methods, not necessarily via known compression algorithms). There are also a number of additional optimizations that are possible due to the unique structure of LWE.

At the recommended security level, Dilithium offers signatures of 2.7KB and public keys of 1.5KB, this is obviously much more than the 32-byte public keys and 64-byte signatures of pre-quantum schemes, but also much better than stateless hash-based signatures. It is good to keep in mind that these schemes are still pretty new, and it is possible that better algorithms will be found to solve the LWE problem, thus potentially increasing the sizes of public keys and signatures, or instead that long-term we will find better techniques to reduce the sizes of these parameters. In general, it is possible that quantum resistance will always come with a cost in size.

14.4 Code-based cryptography and other post-quantum candidates

This is not all there is to post-quantum cryptography. At the time of this writing (2020), the NIST post-quantum cryptography competition has a number of other constructions based on different paradigms. Let me briefly go over some of them.

Code-based cryptography. In 1978, Robert McEliece published the first cryptosystem based on coding theory. At that time, coding theory had only been used to detect, and even correct, errors happening in plaintext transmission or storage. Errors can happen for various reasons, from bugs in the software to cosmic radiations (radiation from space!), and will lead to flipped or lost bits of data (as pictured in figure 14.12). In these cases, especially when authenticated encryption is not used, it is important to detect errors in order to ensure some level of integrity for communicating and storing data.

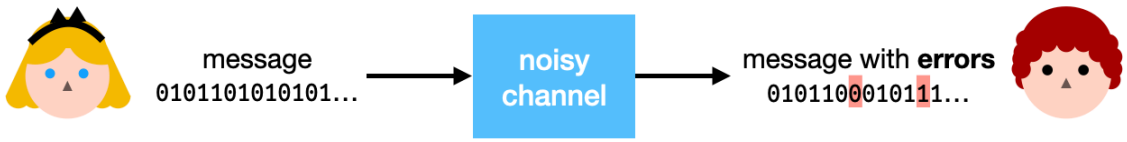


Figure 14.12 Physical noise can happen naturally for all sorts of reasons. This physical phenomenon can impact the integrity of data being transmitted or stored.

Internet uses error-detecting codes on almost all layers, with "checksums" present in ethernet frames, IP packets, and TCP packets. These checksums are just computations similar to hash functions, in that they are required to change if the slightest bit of the data they are associated to is altered. These checksums are not cryptographic in the sense that a malicious man in the middle can easily forge new ones and create collisions or second pre-images, but they are enough to detect up to a specified number of random and non-malicious errors. While some codes allow for

errors to be corrected, in the case of the internet they are only here to detect errors as it is fast to just retransmit a packet that didn't go through correctly.

On the other hand, error-correcting codes work like this: a message is first expanded (in the sense that it is encoded into something that contains more information, which makes the message larger) resulting in what is called a **codeword**. After going through the network, or being stored for a while, the retrieved codeword (with potential errors in it) can go through a decoding algorithm that will be able to correct up to the specified number of errors. I illustrate this in figure 14.13.

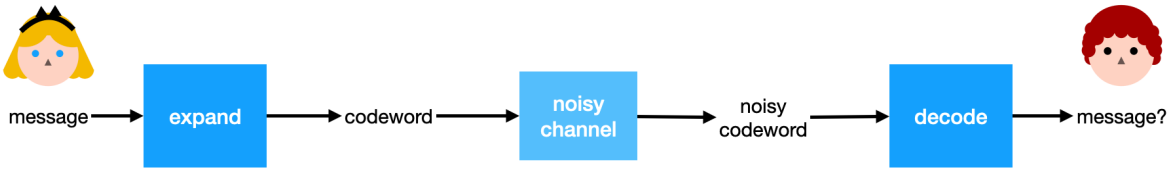


Figure 14.13 To protect against naturally-occuring physical noise altering data in transit or storage, error-correcting codes can be used. Data first needs to be expanded into a codeword, and later that codeword can be decoded to fix a certain number of errors.

Looking at error-correcting code, McEliece figured that there was an asymmetric cryptosystem to be discovered in there. The idea behind McEliece's key encapsulation mechanism is simply put to generate random error-correcting code parameters, and to:

- keep the parts to decode secret
- make the encoding/expanding part public (after some obfuscation/encryption in order to avoid leaking too much information)

This way, anyone can take the expanding code to encode a message, and then add enough errors to the resulting codeword so that attackers who intercept that message wouldn't be able to remove the errors without knowledge of the decoding parts. I illustrate this in figure 14.14.

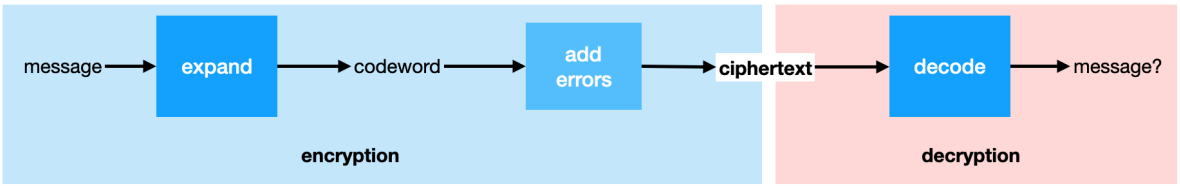


Figure 14.14 McEliece's key encapsulation mechanism simply takes the error-correcting code flow pictured in figure 14.13, and splits it in two. To encrypt a message, expand it into a codeword and add errors manually (emulating a noisy channel); the resulting noisy codeword is the ciphertext. To decrypt, simply decode the noisy codeword to remove all added errors and retrieve the original message.

The described scheme is part of the finalists of NIST's post-quantum cryptography competition under the name **Classic McEliece**. It's probably the submission with the largest team (around 20 cryptographers) and is considered solid considering the 40 years of cryptanalysis that McEliece's scheme has withstood.

Isogeny-based cryptography. While we have groups formed of elliptic curve points, the isogeny-based cryptography field takes this even further, and creates groups based of elliptic curves themselves. There, an isogeny is a function that maps not a point to another point (like scalar multiplication), but a curve to another curve! There is one public-key encryption scheme based on this paradigm in the competition, called **SIKE**. The main advantage of such schemes are parameter sizes: keys and ciphertexts are only hundreds of bytes.

Symmetric key-based cryptography. Interestingly, among the submissions to the NIST post-quantum cryptography competition lies an outsider called **Picnic**, based solely on block ciphers and hash functions. It is thus somewhat similar to hash-based signatures, in the sense that it only relies on well-studied primitives and thus provides strong assurances out of the box. Picnic is also special in that it relies on secure multi-party computation and zero-knowledge proofs, two primitives that can be instantiated with symmetric cryptographic primitives only, and that I will talk about in chapter 15.

Multivariate cryptography. Multivariate cryptography is based on multivariate polynomials, which are just a term for polynomials that have multiple unknowns or variables. For example, $x^2 + 3x + y + 4y^3 = 0$ is a multivariate polynomials since it has more than one variable: x and y. Cryptosystems based on such equations use a much larger number of variables, and in several polynomials, to represent public keys. The **Rainbow** signature scheme is the only multivariate submission still standing in the NIST post-quantum cryptography competition (as of 2020). Rainbow is based on the unbalanced oil and vinegar scheme, which signs messages by taking the message as output of the several multivariate polynomials that form the public key, and finds a valid set of input values as pre-image; that pre-image is a signature, and can be verified by simply evaluating the polynomials at these values and making sure that the output is the message associated to the signature. The security of such a scheme relies on the fact that it is hard to find the pre-image of a set of multivariate equations. In order to make it work the polynomials used to compute a signature, and to verify a signature, are slightly different: one is easy to invert, while the other is not.

Unfortunately, the NIST post-quantum cryptography competition is not finished at the time of this writing. The NIST has announced that an initial standard will be published in 2022, but I expect that the field will continue to move, at least until post-quantum computers are shown to be a real threat. So while there's still a lot of unknowns, it also means that there's a lot of exciting room for research. If this is interesting to you, I recommend taking a look at NIST reports at https://nist.gov/pqcrypto.

14.5 Do I need to panic?

To summarize, quantum computers are a huge deal for cryptography if they do realize themselves. So what's the take away here? Do you need to throw everything you're doing and transition to post-quantum algorithms? Well… it's not that simple.

Ask any expert and you'll receive different kinds of answers. For some, it's 5 to 50 years away, for others it'll never happen. Michele Mosca, the director of the Institute for Quantum Computing, estimated "a 1/7 chance of breaking RSA-2048 by 2026 and a 1/2 chance by 2031"; while Mikhail Dyakonov, a researcher at the CNRS in France, stated publicly "Could we ever learn to control the more than 10^{300} continuously variable parameters defining the quantum state of such a system? My answer is simple. No, never." While physicists—not cryptographers—know better, they can still be incentivized to hype their own research in order to get funding. As I am no physicist, I will simply say that we should remain skeptical of extraordinary claims, while preparing for the worst.

The question is not "does it work?", but rather "will it scale?"

There exist many many challenges for scalable quantum computers (those who can destroy cryptography) to become a reality; the biggest ones seem to be about the amount of noise and errors that is difficult to reduce or correct. Scott Aaronson, a computer scientist at the University of Texas, puts it as "You're trying to build a ship that remains the same ship, even as every plank in it rots and has to be replaced."

In any case, if you're really worried, and the confidentiality of your assets needs to hold for long periods of time, it is not crazy and relatively easy to increase the parameters of every symmetric cryptographic algorithm you're using. That being said, if you're doing a key exchange in order to obtain an AES-256-GCM key, that asymmetric cryptography part is still vulnerable to quantum computers and protecting the symmetric cryptography alone won't be enough.

For asymmetric cryptography, it is too early to really know what is safe to use. Best wait for the end of the NIST competition in order to obtain more cryptanalysis, and in turn more confidence, in these novel algorithms.

> *At present, there are several post-quantum cryptosystems that have been proposed, including lattice-based cryptosystems, code-based cryptosystems, multivariate cryptosystems, hash-based signatures, and others. However, for most of these proposals, further research is needed in order to gain more confidence in their security (particularly against adversaries with quantum computers) and to improve their performance.*
>
> *– NIST Post-Quantum Cryptography Call for Proposals (2017)*

If you're too impatient, and can't wait for the result of the NIST competition, one thing you can do is to use both a current scheme **and** a post-quantum scheme in your protocol. For example,

you could "cross-sign" messages using Ed25519 and Dilithium, or in other words attach a message with two signatures from two different signature schemes. If it turns out that Dilithium is broken, an attacker would still have to break Ed25519, and if it turns out that quantum computers are real, then the attacker would still have the Dilithium signature that they can't forge.

> **NOTE** This is what Google did in 2018, and then again in 2019 with Cloudflare, experimenting with a hybrid key exchange scheme in TLS connections between a small percentage of Google Chrome users and servers from both Google and Cloudflare. The hybrid scheme was a mix of X25519 and one post-quantum key exchange (New Hope in 2018, HRSS and SIKE in 2019), where both the output of the current key exchange and the post-quantum key exchange were mixed together into HKDF to produce one single shared secret.

And finally, I will reemphasis that hash-based signatures are well-studied and well-understood. Even though they present some overhead, schemes like XMSS and SPHINCS+ can be used today, and XMSS even has ready-to-be-used standards.[57]

14.6 Summary

- Quantum computers are based on quantum physics and can provide a non-negligible speed up for specific computations.
- Not all algorithms can run on a quantum computer, and not all algorithms can compete with a classical computer. Two notable algorithms that worry cryptographers are:
  - Shor's algorithm, which can efficiently solve the discrete logarithm problem and the factorization problem. It breaks most of today's asymmetric cryptography.
  - Grover's algorithm, which can efficiently search for a key or value in a space of 2^{128} values, this impacts most symmetric algorithms with 128-bit security, but boosting a symmetric algorithm's parameters to provide 256-bit security is enough to thwart quantum attacks.
- The field of post-quantum cryptography aims at finding novel cryptographic algorithms to replace today's asymmetric cryptographic primitives like asymmetric encryption, key exchanges, and digital signatures.
- NIST started a post-quantum cryptography standardization effort in 2016. There are currently 7 finalists and the effort is now entering its final round of selection.
- Hash-based signatures are signature schemes that are only based on hash functions. The two main standards are XMSS, a stateful signature scheme, and SPHINCS+, a stateless signature scheme.
- Lattice-based cryptography is promising, as it provides shorter keys and signatures. Two of the most promising candidates there are based on the learning with errors problem: Kyber is an asymmetric encryption and key exchange primitive, Dilithium is a signature scheme.
- Other post-quantum schemes exist, and are being proposed as part of the NIST post-quantum cryptography competition. They include schemes based on code theory, isogenies, symmetric-key cryptography, and multivariate polynomials. NIST's competition is scheduled to end in 2022, which still leaves a lot of room for new attacks or optimizations to be discovered.
- It is not clear when quantum computers will be efficient enough to destroy cryptography, or if it is possible for them to get there.
- If you have requirements to protect data for a long period of time, you should consider transitioning to post-quantum cryptography:
  - Upgrade all usage of symmetric cryptographic algorithms to use parameters that provide 256-bit security (for example, move from AES-128-GCM to AES-256-GCM, and from SHA-3-256 to SHA-3-512).
  - Use hybrid schemes that mixes post-quantum algorithms with pre-quantum algorithms. For example, always sign messages with both Ed25519 and Dilithium, or always perform a key exchange with both X25519 and Kyber (deriving a shared secret from the two key exchange outputs obtained).
  - Use hash-based signatures like XMSS and SPHINCS+, which are well-studied and well-understood. XMSS has the advantage of having already been standardized and approved by the NIST.

15 Is this it? Next-generation cryptography

This chapter covers

- How cryptography can help get rid of trusted third parties via secure multi-party computation (MPC).
- How untrusted third parties can still compute useful programs on encrypted data via fully homomorphic encryption (FHE).
- How zero-knowledge proofs (ZKPs) can be used to hide parts of a program's execution to compress or add confidentiality to a protocol.

I started this book with the idea that readers who would get through most of the chapters would also be interested in the future of real-world cryptography. While you're reading an applied and practical book, with a focus on what is in use today, the field of cryptography is rapidly changing (as we've seen in recent years with cryptocurrencies, for example). A number of theoretical cryptographic primitives and protocols are making their ways into the applied cryptography world. Maybe because these theoretical primitives are finally finding a use case, or maybe because they're finally becoming efficient enough to be used in real-world applications. Whatever the reason, the real-world of cryptography is definitely growing and getting more exciting. So in this section, to leave you with a taste of what the future of real-world cryptography might look like (perhaps in the next 10 to 20 years), I will briefly survey three primitives:

- **Secure multi-party computation (MPC)**. This is the subfield of cryptography that allows different participants to execute a program together, without necessarily revealing their own input to the program.
- **Fully Homomorphic Encryption (FHE)**. The holy grail of cryptography, a primitive used to allow arbitrary computations on encrypted data.
- **General-purpose zero-knowledge proofs (ZKP)**. The primitive you've learned about in chapter 7 which allows you to prove that you know something without revealing the

something, but this time applied more generally to arbitrary programs.

This chapter contains the most advanced and complicated concepts in the book. For this reason, I recommend that you glance at it, and move on to chapter 16 to read the conclusion. Then, if you are motivated to learn more about the inners of these more advanced concepts, come back to this chapter.

Let's get started!

15.1 The more the merrier: secure multi-party computation (MPC)

Secure multi-party computation (MPC) is a field of cryptography that came into existence in 1982 with the famous millionaires' problem.[58] Simply put, MPC is a way for multiple participants to compute a program together. But before learning more about this new primitive, let's see why it's useful.

We know that with the help of a trusted third party, any distributed computation can easily be computed. This trusted third party can perhaps maintain the privacy of each participant's input, and maybe restrict the amount of information revealed by the computation to specific participants. In the real world though, we don't like trusted third-parties too much, we know that they are pretty hard to come by and they don't always respect their commitment. MPC allows us to completely remove trusted third parties from a distributed computation and enables participants to compute the computation by themselves, without revealing their respective inputs to one another, through a cryptographic protocol. With that in mind, using MPC in a system is pretty much the equivalent to using a trusted third party (see figure 15.1).

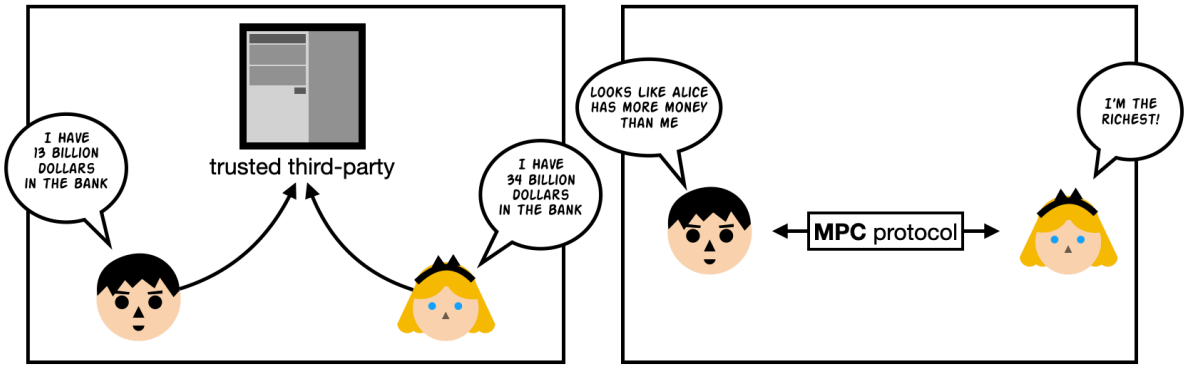


Figure 15.1 A secure multi-party computation (MPC) protocol allows a distributed computation that can be computed via a trusted third party (image on the left) to be computed without the need of a trusted third party (image on the right).

Note that you've already seen some MPC protocols! Threshold signatures and distributed key generations, covered in chapter 8, are examples of MPC. More specifically, these examples are

part of a subfield of MPC called **threshold cryptography**, which has been receiving a lot of love in more recent years with, for example, the NIST in mid-2019 kicking off a standardization process for threshold cryptography.

15.1.1 Private set intersection (PSI)

Another well-known subfield of MPC is the field of **private set intersection (PSI)** which poses the following problem: Alice and Bob have a list of words, and they want to know which words (or perhaps just how many) they have in common without revealing their respective list of words.

One way to solve this problem is to use the oblivious PRF (OPFR) construction you learned about in chapter 11:

1. Bob generates a key for the OPRF.
2. Alice obtains the random values *PRF(key, word)* for every word in her list, using the OPRF protocol (so she doesn't learn the PRF key, and Bob doesn't learn the words).
3. Bob then can compute the list of *PRF(key, word)* for his own words and send it to Alice, who is then able to compare if any PRF of Bob's outputs matches her own PRF outputs.

I illustrate this protocol in figure 15.2.

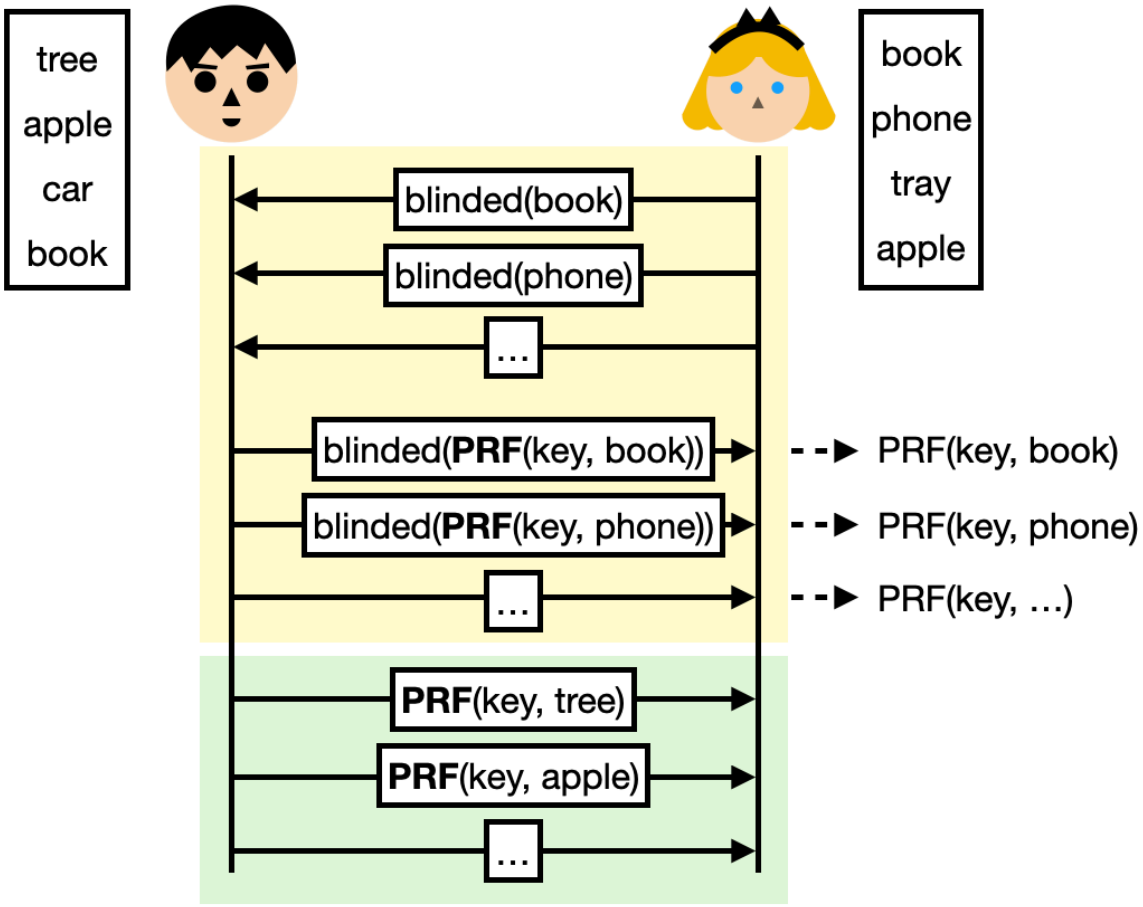


Figure 15.2 Private set intersection (PSI) allows Alice to learn what words she has in common with Bob. First, she blinds every word she has in her list, and uses the OPRF protocol with Bob to obtain the PRF of Bob's key with her words. Finally, Bob sends her the PRF of his key with his words. Alice can then see if anything matches to learn what words they have in common.

PSI is a promising field which is starting to see more and more adoption as it has shown to be more and more practical. For example Google's "password checkup" integrated into the Chrome browser uses PSI to warn you when some of your passwords have been detected in password dumps (following password breaches), without actually seeing your passwords. Interestingly, Microsoft also does this for its Edge browser, but uses fully homomorphic encryption (which I'll be introducing in the next section) to perform the private set intersection.

On the other hand, the developers of the Signal messaging application (discussed in chapter 10) decided that PSI was still too slow to perform contact discovery (in order to figure out who you can talk to based on your phone's contact list) and instead uses SGX (covered in chapter 13) as a trusted third party.

15.1.2 General-purpose MPC

But more generally, MPC has many different solutions aiming at the computation of arbitrary programs. General-purpose MPC solutions all provide different types of properties (for example, how many dishonest participants can the protocol tolerate? Are participants malicious or just honest but curious?[59] Is it fair to all participants if some of them terminate the protocol early? And so on.) or levels of efficiency (from hours to milliseconds).

Before a program can be securely computed with MPC, it needs to be translated into an **arithmetic circuit**. Arithmetic circuits are successions of additions and multiplications, and since they are Turing complete, they can represent **any** program! For an illustration of an arithmetic circuit, see figure 15.3.

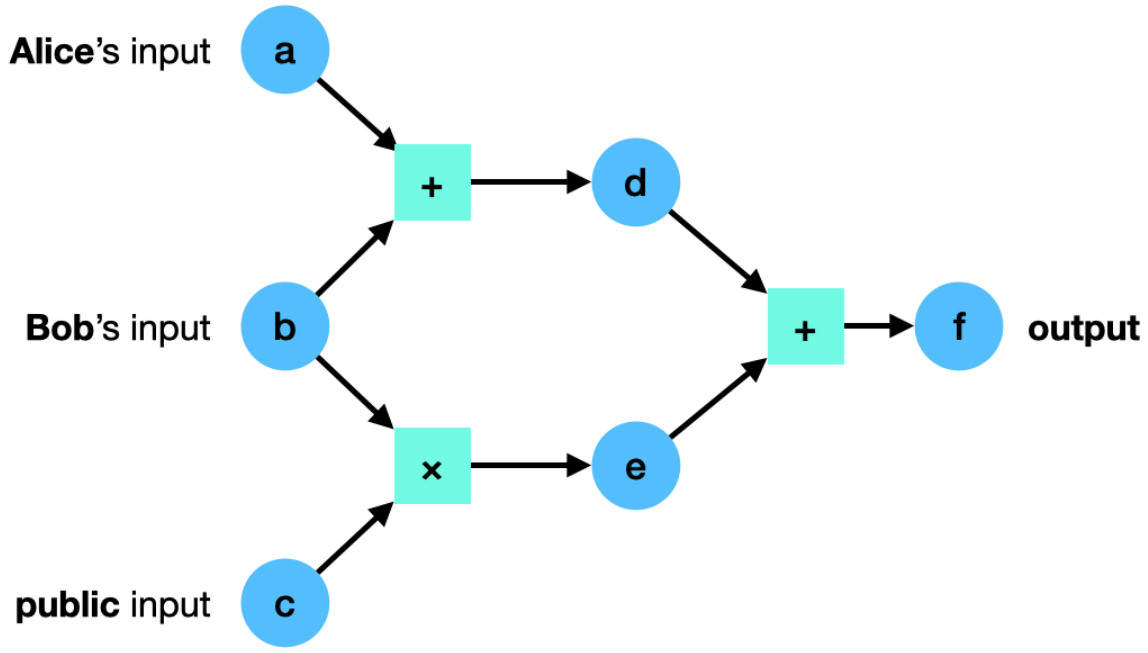


Figure 15.3 An arithmetic circuit is a number of (addition or multiplication)gates linking inputs to outputs. In the diagram values travel from left to right. For example, d = a + b. Here, the circuit only outputs one value f = a+b+bc, but it can in theory have multiple output values. Notice that different inputs to the circuit are provided by different participants, or can be public (known to everyone).

Before taking a look at the next primitive, let me give you a simplified example of an (honest-majority) general-purpose MPC built via Shamir's secret sharing. There exist many more schemes, but this one is simple enough to fit here in a three-step explanation: share enough information on each input in the circuit, evaluate every gate in the circuit, reconstruct the output. Let's see each step in more detail.

The **first step** is for every participant to have enough information about **each input of the circuit**. Public inputs are shared publicly, while private inputs are shared via Shamir's secret

sharing (covered in chapter 8). I illustrate this in figure 15.4.

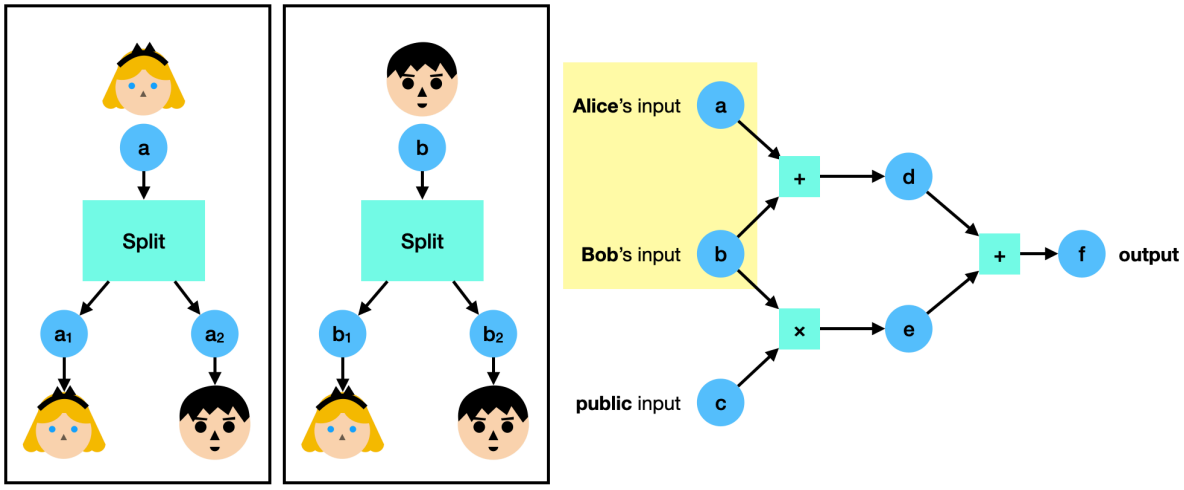


Figure 15.4 The first step of a general-purpose MPC with secret sharing is to have participants split their respective secret inputs (using Shamir's secret sharing scheme) and distribute the parts to all participants. For example, here Alice splits her input a into a_1 and a_2. Since there are only two participants in this example, she gives the first share to herself, and gives Bob the second one.

The **second step** is to **evaluate every gate of the circuit**. For some technical reasons I'll omit here, addition gates can be computed locally, while multiplication gates have to be computed interactively (participants must exchange some messages). For an addition gate, simply add the input shares you have; for a multiplication gate, multiply the input shares. What you get is a share of the result, as illustrated in figure 15.5. At this point the shares can be exchanged in order to reconstruct the output, or kept separately to continue the computation if they represent an intermediate value.

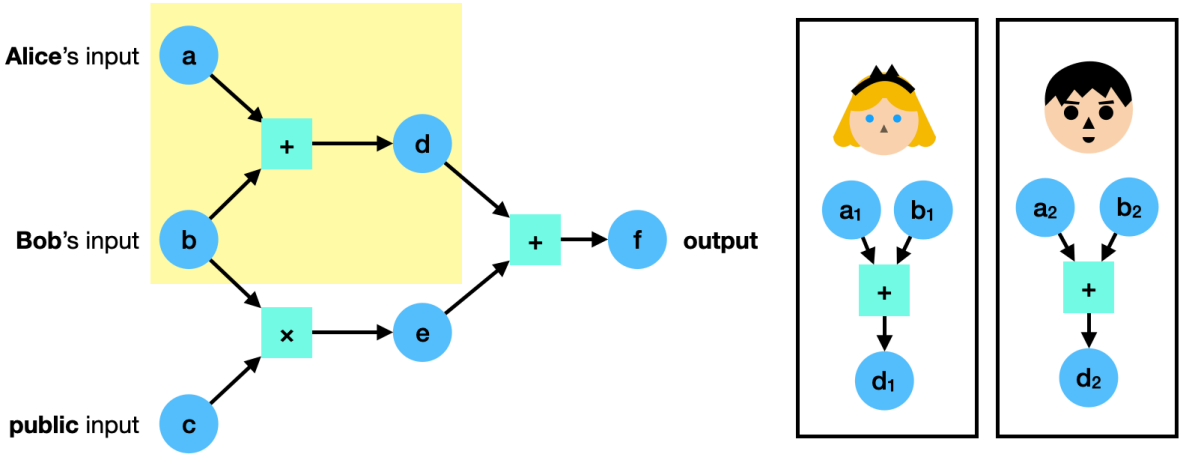


Figure 15.5 The second step of a general-purpose MPC with secret sharing is to have participants compute every single gate in the circuit. For example, a participant can compute an addition gate by adding the two input Shamir shares that they have, which produces a Shamir share of the output.

The **final step** is to **reconstruct the output**. At this point, the participants should all own a share of the output, which they can use to reconstruct the final output using the final step of Shamir's secret sharing scheme.

15.1.3 The state of MPC

There's been huge progress in the last decade to make MPC practical. It is a field of many different use cases and one should be on the lookout for the potential applications that can benefit from this new-ish primitive. Note that unfortunately no real standardization effort exists, and while several MPC implementations can be considered practical for many use cases today, they are not easy to use.

By the way, the general-purpose MPC construction I explained in this section is based on secret sharing, but there are more ways to construct MPC protocols. A well-known alternative is known as **garbled circuits**, a type of construction first proposed by Yao in his 1982 paper introducing MPC, another one is based on **fully homomorphic encryption** which is what you are going to learn about in the next section!

15.2 Fully homomorphic encryption (FHE) and the promises of an encrypted cloud

For a very long time in cryptography, a question has troubled many cryptographers: **is it possible to compute arbitrary programs on encrypted data?** Imagine that you could encrypt the values a, b, and c separately, send the ciphertexts to a service, and ask that service to return the encryption of $a \times 3b + 2c + 3$ which you could then decrypt. The important idea here is that the service never learns about your values and always deals with ciphertexts. Of course this calculation might not be too useful, but with additions and multiplications one can compute actual programs on the encrypted data. This interesting concept, originally proposed in 1978 by Rivest, Adleman, and Dertouzos, is what we call **fully homomorphic encryption (FHE)** (or also, as it used to be called, the *holy grail of cryptography*). I illustrate this cryptographic primitive in figure 15.6.

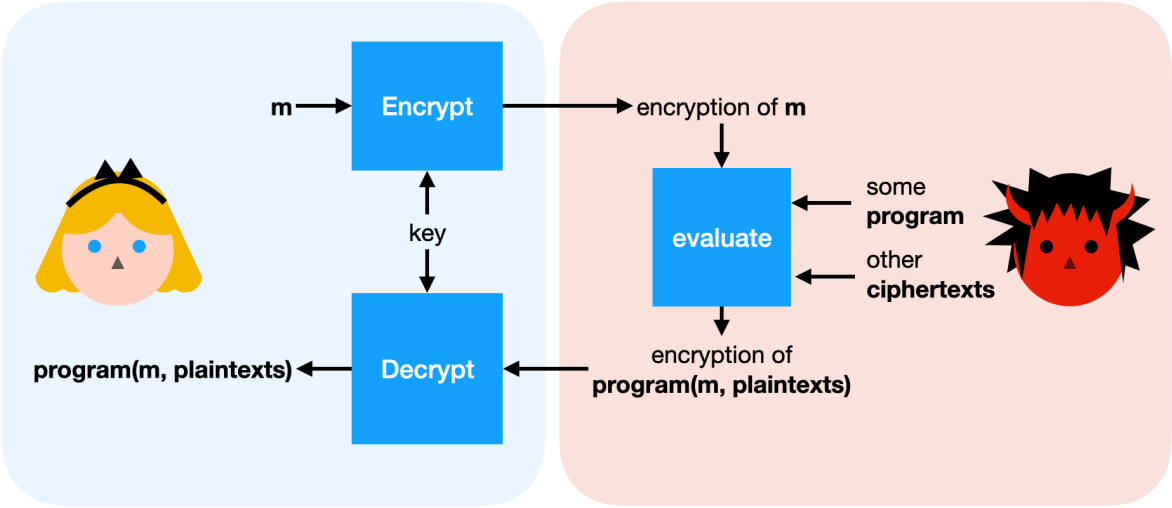


Figure 15.6 Fully homomorphic encryption (FHE) is an encryption scheme that allows for arbitrary computations over encrypted content. Only the owner of the key can decrypt the result of the computation.

15.2.1 An example of homomorphic encryption with RSA encryption

By the way, you've already seen some cryptographic schemes that should make you feel like you know what I'm talking about. Think of RSA (covered in chapter 6): given a $ciphertext = message^e \bmod N$, someone can easily compute some restricted function of the ciphertext: $n^e \times ciphertext = (n \times message)^e \bmod N$ for any number n they want (although it can't be too big), which will decrypt to $n \times message$.

Of course, this is not a desired behavior for RSA, and it has led to some attacks (for example, Bleichenbacher's attack mentioned in chapter 6). So in practice RSA breaks the usefulness of its "homomorphism" property by using a padding scheme. Note that RSA is homomorphic only for the multiplication, which is not enough to compute arbitrary functions (both multiplication and addition are needed). Due to this limitation, we say that RSA is **partially homomorphic**.

15.2.2 The different types of homomorphic encryption

Other type of homomorphic encryptions are:

- **Somewhat homomorphic**, which means partially homomorphic for one operation (addition or multiplication), and homomorphic for the other operation in limited ways. For example, additions are unlimited up to a certain number but only a few multiplications can be done.
- **Leveled homomorphic**, which means both addition and multiplication are unlimited up to a certain number.
- **Fully homomorphic**, which is the real deal: addition and multiplication are unlimited.

Before the invention of FHE, several types of homomorphic encryption schemes were proposed, but none could achieve what fully homomorphic encryption promised. The reason is that by

evaluating circuits on encrypted data, some **noise** grows; after a point, the noise has reached a threshold that makes decryption impossible. And for many years, some researchers tried to prove that perhaps there was some information theory result that could show that fully homomorphic encryption was impossible. That is, until it was shown to be possible…

15.2.3 Bootstrapping, the key to fully homomorphic encryption

> *One night, Alice dreams of immense riches, caverns piled high with silver, gold and diamonds. Then, a giant dragon devours the riches and begins to eat its own tail! She awakes with a feeling of peace. As she tries to make sense of her dream, she realizes that she has the solution to her problem.*
>
> *– Craig Gentry Computing Arbitrary Functions of Encrypted Data (2009)*

In 2009, Craig Gentry, a PhD student of Dan Boneh, proposed the first-ever fully homomorphic encryption construction. Gentry's solution was called **bootstrapping**, which in effect was to evaluate a decryption circuit on the ciphertext, every so often, in order to reduce the noise to a manageable threshold. Interestingly, the decryption circuit itself does not reveal the private key and can be computed by the untrusted party. Bootstrapping allowed turning a leveled FHE scheme into an FHE scheme. Gentry's construction was slow, and quite impractical, reporting about 30 minutes per basic bit operation, but as with any breakthrough it only got better with time. It also showed that fully homomorphic encryption was possible.

How does bootstrapping work? Let's take a look at some intuition. First, I need to mention that we'll need not a symmetric encryption system, but a public-key encryption system where a public key can be used to encrypt and a private key can be used to decrypt. Now, imagine that after executing a certain number of additions and multiplications on a ciphertext, you reach a level of noise that still allows you to decrypt the ciphertext correctly, that is unless you perform one more homomorphic operation. I illustrate this in figure 15.7.

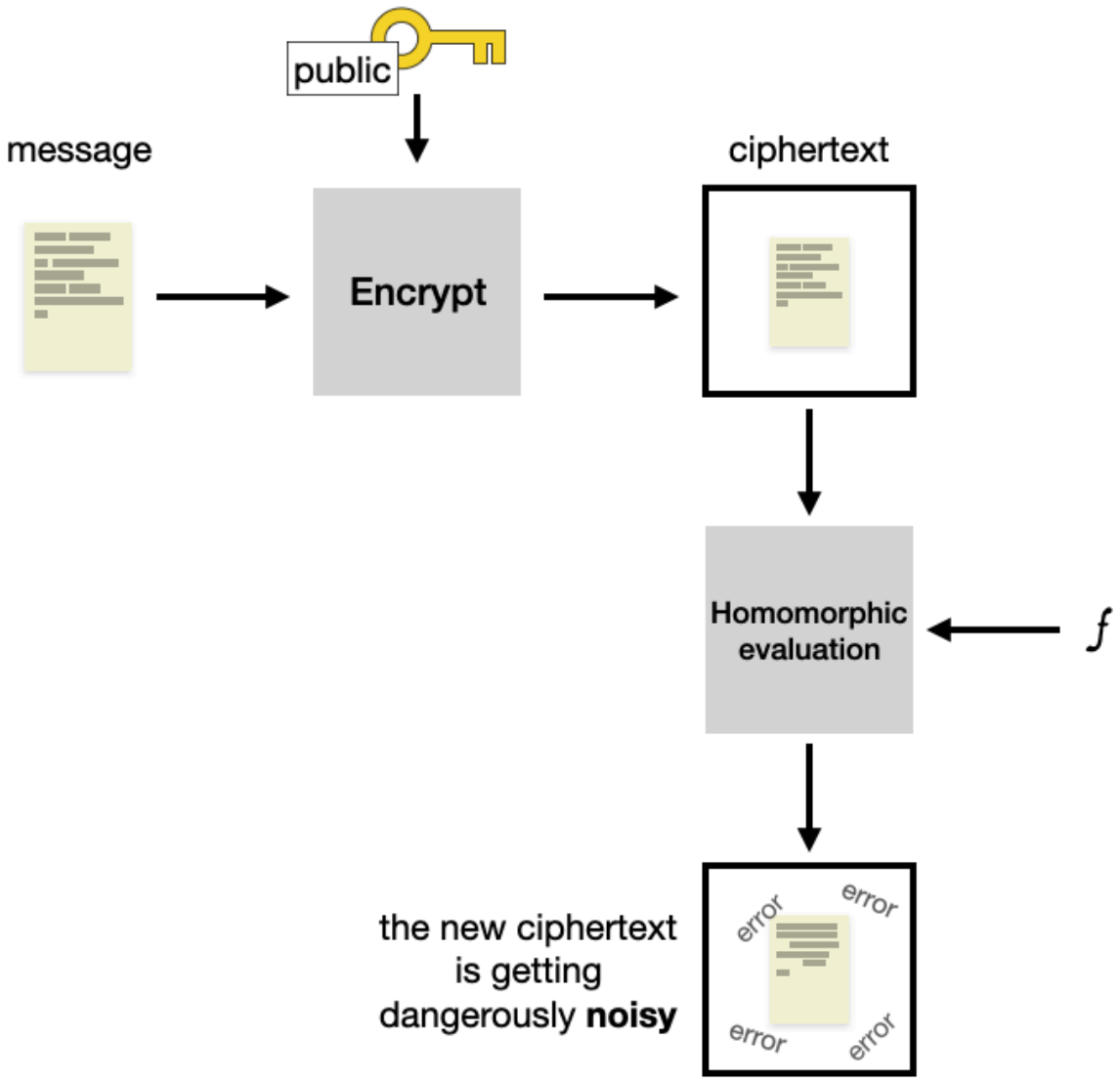


Figure 15.7 After encrypting a message with a fully homomorphic encryption algorithm, operating on it will increase its noise to dangerous thresholds (where decryption can become impossible).

You could think that you're stuck, but bootstrapping unstucks you by removing the noise out of that ciphertext. To do that, you re-encrypt the noisy ciphertext under another public key (usually called the **bootstrapping key**) to obtain an encryption of that noisy ciphertext. Notice that the new ciphertext has no noise. I illustrate this in figure 15.8.

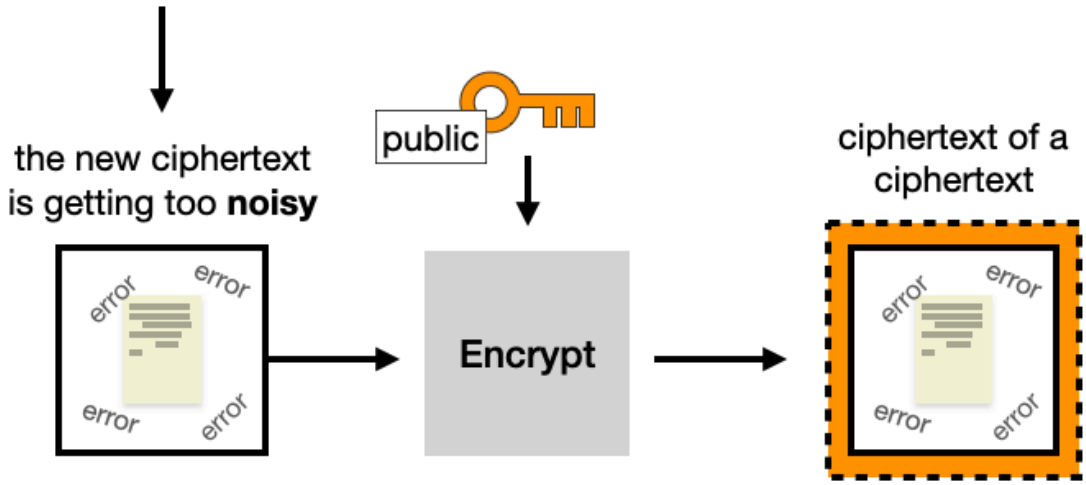


Figure 15.8 Building on figure 15.7, to eliminate the noise of the ciphertext you can decrypt it. But since you don't have the secret key, instead you re-encrypt the noisy ciphertext under another public key (called the bootstrapping key) to obtain a new ciphertext without error (of a noisy ciphertext).

Now comes the magic: you are provided with the initial private key, not in clear, but encrypted under that bootstrapping key. This means that you can use it, with a decryption circuit, to homomorphically decrypt the inner noisy ciphertext. If the decryption circuit produces an acceptable amount of noise then it will work, and you will end up with the result of the first homomorphic operation encrypted under the bootstrapping key. I illustrate this in figure 15.9.

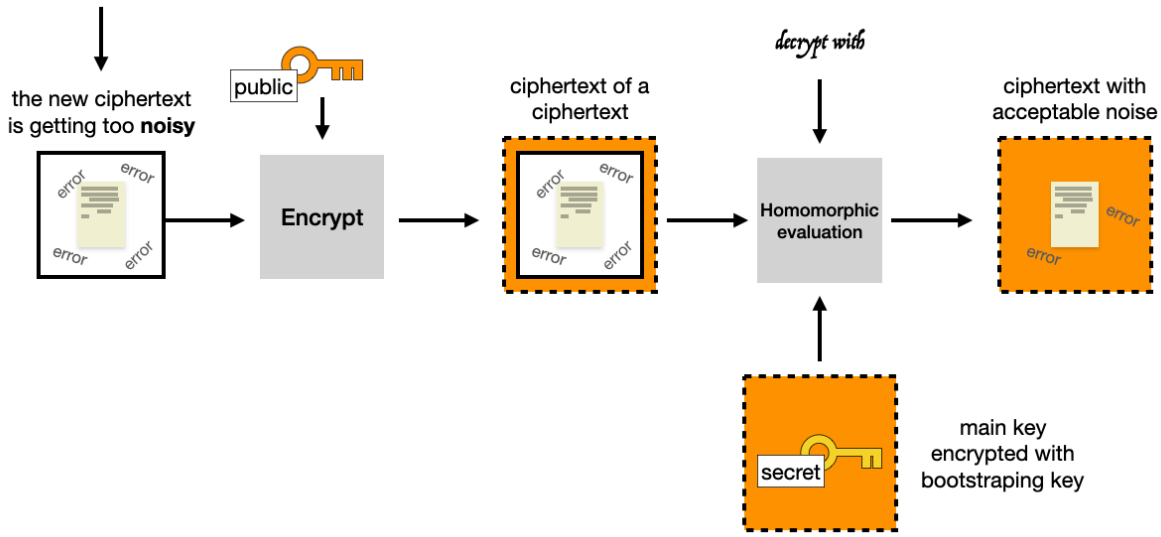


Figure 15.9 Building on figure 15.9, you finally use the initial secret key encrypted to the bootstrapping key, to apply the decryption circuit to that new ciphertext. This will effectively decrypt the noisy ciphertext in place, removing the errors. There will be some amount of errors due to the decryption circuit.

If the remaining amount of errors allows you to do **at least** one more homomorphic operation (+

or ×), then you are gold, you have a fully homomorphic encryption algorithm (ince you can always, in practice, run the bootstrapping after or before every operation).

Note that you can set the bootstrapping keypair to be the same as the initial keypair. It's a bit weird, you get some circular security oddity, but it seems to work and no security issues are known.

15.2.4 An FHE scheme based on the learning with errors problem

Before moving on, let's see one example of an FHE scheme based on the learning with errors problem we've seen in chapter 14. I'll explain a simplified version of the GSW scheme (named after the last names of the authors Craig Gentry, Amit Sahai, and Brent Waters).

To keep things simple, I'll introduce a secret-key version of the algorithm, but just keep in mind that it is relatively straightforward to transform such a scheme into a public-key variant (which we need for bootstrapping).

Take a look at the following equation where $\boldsymbol{C}$ is a square matrix, $\boldsymbol{s}$ is a vector, and m is a scalar (a number):

$$\boldsymbol{Cs} = m\boldsymbol{s}$$

In this equation, $\boldsymbol{s}$ is called an *eigenvector* and m an *eigenvalue*. If these words are foreign to you, don't worry about it, they don't matter much here.

The first intuition in our FHE scheme is obtained by looking at these eigenvectors and eigenvalues. The observation is that if we set m to a single bit we want to encrypt, $\boldsymbol{C}$ to be the ciphertext, and $\boldsymbol{s}$ to be the secret key, then we have an (insecure) homomorphic encryption scheme to encrypt one bit. (Of course, we assume there is a way to obtain a random ciphertext $\boldsymbol{C}$ from a fixed bit m and a fixed secret key $\boldsymbol{s}$.) I illustrate this in figure 15.10 in a lego kind of way.

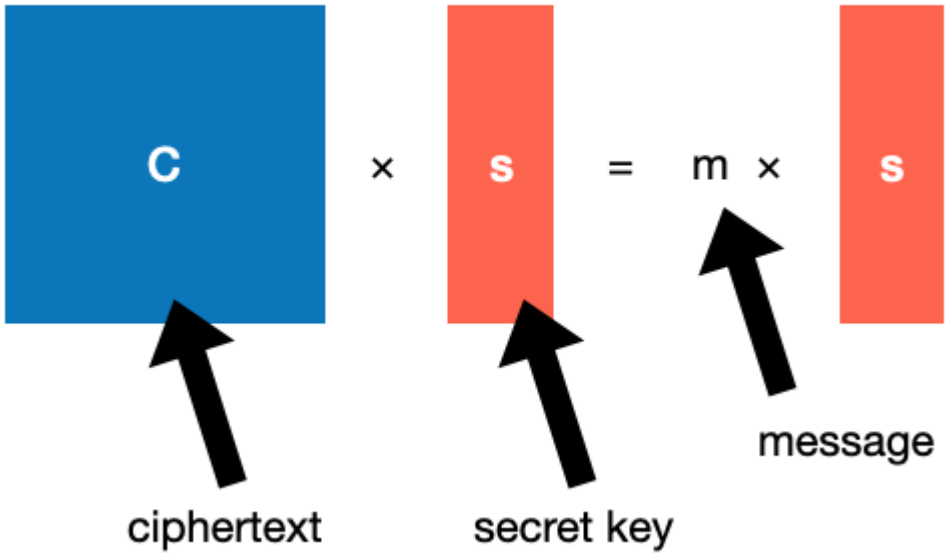


Figure 15.10 We can produce an insecure homomorphic encryption scheme to encrypt a single bit m with a secret vector s by interpreting m as an eigenvalue and s as an eigenvector and finding the associated matrix C (which will be the ciphertext).

To decrypt a ciphertext, you would thus multiply the matrix with the secret vector $\boldsymbol{s}$ and see if you obtain the secret vector back, or 0.

You can verify that the scheme is fully homomorphic by checking that the decryption of two ciphertexts added together ($\boldsymbol{C}_1 + \boldsymbol{C}_2$) results in the associated bits added together:

$$(\boldsymbol{C}_1 + \boldsymbol{C}_2)\boldsymbol{s} = \boldsymbol{C}_1\boldsymbol{s} + \boldsymbol{C}_2\boldsymbol{s} = b_1\boldsymbol{s} + b_2\boldsymbol{s} = (b_1 + b_2)\boldsymbol{s}$$

And that the decryption of two ciphertexts multiplied together ($\boldsymbol{C}_1 \times \boldsymbol{C}_2$) results in the associated bits multiplied together:

$$(\boldsymbol{C}_1 \times \boldsymbol{C}_2)\boldsymbol{s} = \boldsymbol{C}_1 (\boldsymbol{C}_2\boldsymbol{s}) = \boldsymbol{C}_1 (b_2\boldsymbol{s}) = b_2 \boldsymbol{C}_1\boldsymbol{s} = (b_2 \times b_1) \boldsymbol{s}$$

Unfortunately that scheme is insecure as it is trivial to retrieve the eigenvector (the secret vector $\boldsymbol{s}$) from $\boldsymbol{C}$.

What about adding a bit of noise? We can change this equation a bit to make it look like our learning with errors problem:

$$\boldsymbol{C}\boldsymbol{s} = m\boldsymbol{s} + \boldsymbol{e}$$

This should look more familiar.

Again, we can verify that the addition is still homomorphic:

$$(\boldsymbol{C}_1 + \boldsymbol{C}_2)\boldsymbol{s} = \boldsymbol{C}_1\boldsymbol{s} + \boldsymbol{C}_2\boldsymbol{s} = b_1\boldsymbol{s} + \boldsymbol{e}_1 + b_2\boldsymbol{s} + \boldsymbol{e}_2 = (b_1 + b_2)\boldsymbol{s} + (\boldsymbol{e}_1 + \boldsymbol{e}_2)$$

Here notice that the error is growing $(\boldsymbol{e}_1 + \boldsymbol{e}_2)$ which is what we expected. We can also verify that the multiplication is still working as well:

$$(\boldsymbol{C}_1 \times \boldsymbol{C}_2)\boldsymbol{s} = \boldsymbol{C}_1 (\boldsymbol{C}_2\boldsymbol{s}) = \boldsymbol{C}_1 (b_2\boldsymbol{s} + \boldsymbol{e}_2) = b_2 \boldsymbol{C}_1\boldsymbol{s} + \boldsymbol{C}_1 \boldsymbol{e}_2 = b_2 (b_1\boldsymbol{s} + \boldsymbol{e}_1) + \boldsymbol{C}_1 \boldsymbol{e}_2 = (b_2 \times b_1)\, \boldsymbol{s} + b_2 \boldsymbol{e}_1 + \boldsymbol{C}_1 \boldsymbol{e}_2$$

Here, $b_2 \boldsymbol{e}_1$ is small (as it is either $\boldsymbol{e}_1$ or 0), but $\boldsymbol{C}_1 \boldsymbol{e}_2$ is potentially large… This is obviously a problem, which I'm going to ignore here to avoid digging too much into the details. If you're interested to learn more, make sure to read Shai Halevi's Homomorphic Encryption report (2017) which does an excellent job at explaining all of these things, and more.

Next, let's take a look at a primitive that is very much in its process of becoming a real-world cryptographic primitive…

15.2.5 Where is it used?

The most touted use case of FHE has always been the cloud: what if I could continue storing my data in the cloud without having them being able to see it? And additionally, what if the cloud could provide useful computations on that encrypted data? And indeed, one can think of many applications where this new primitive could be useful. For example, a spam detector could scan your emails without looking at them, genetic research could be performed on your DNA without actually having to store and protect your privacy-sensitive human code, a database could be stored encrypted and queried on the server side without revealing any data. Yet Phillip Rogaway, in his seminal 2015 paper on "The Moral Character of Cryptographic Work," notes that "FHE […] have engendered a new wave of exuberance. In grant proposals, media interviews, and talks, leading theorists speak of FHE […] as game-changing indications of where we have come. Nobody seems to emphasize just how speculative it is that any of this will ever have any impact on practice."

While Rogaway is not wrong—FHE is still quite slow—advances in the field have been very exciting. At the time of this writing (2020) operations are about one billion times slower than normal operations, yet since 2009 there has been a 10^9 speedup. We are undoubtedly moving towards a future where FHE will be possible for at least some limited applications. Furthermore, not every application needs the full-blown primitive: somewhat homomorphic encryption can also be used in a wide-range of applications and is much more efficient than FHE.

A good indicator that a theoretical cryptography primitive is entering the real world is standardization, and indeed FHE is no foreigner to that with the https://homomorphicencryption.org standardization effort which includes many large companies and universities.

It is still unclear exactly when, where, and in what form homomorphic encryption will make its entry in the real world. What's clear is that it will happen, so stay tuned!

Before going to the next topic, I want to give you some intuition about the current state of the art in FHE. It turns out that some of the best schemes are based on the learning with errors problem you learned about in chapter 14.

15.3 General-purpose zero-knowledge proofs

I talked about zero-knowledge proofs (ZKPs) in chapter 7 on signatures. There, I pointed out that signatures are similar to non-interactive zero-knowledge proofs of knowledge of discrete logarithms. These kinds of ZKPs were invented in the mid 80s by Professors Shafi Goldwasser, Silvio Micali, and Charles Rackoff. Shortly after, Goldreich, Micali, and Wigderson found that we could prove much more than just discrete logarithms or other types of hard problems: we could prove that **any** program had executed correctly even if we removed some of its inputs or outputs (see figure 15.11 for an example). This section focuses on this general-purpose type of ZKP.

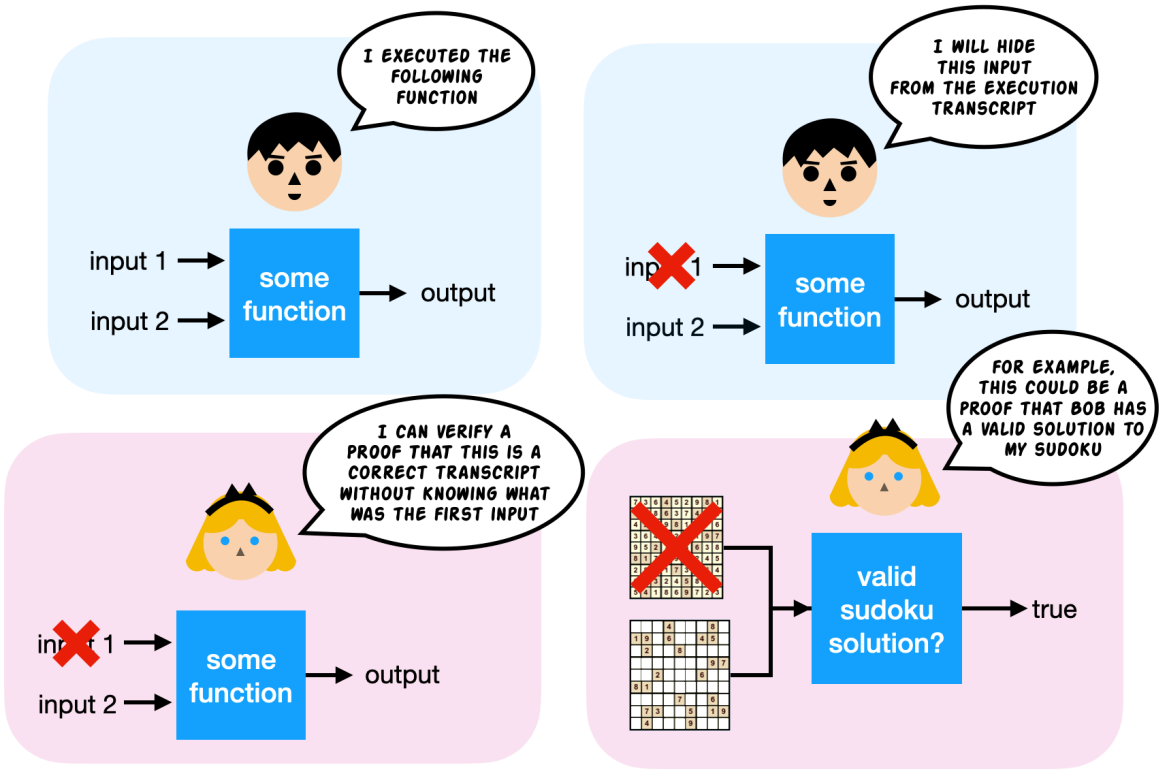


Figure 15.11 General-purpose zero-knowledge proofs allow a prover to convince a verifier about the integrity of an execution trace (the inputs of a program and the outputs obtained after its execution) while hiding some of the inputs or outputs involved in the computation. An example of this is a prover trying to prove that a sudoku can be solved.

ZKP as a field has grown tremendously since its early years. One major reason for this growth

has been the cryptocurrency boom, and their need to provide more confidentiality to transactions on-chain as well as optimize on space. The field of ZKP is still growing extremely fast as of the time of this writing, and it is quite hard to follow what are all the modern schemes that exist and what types of general-purpose ZKPs there are.

Fortunately for us, this problem was getting large enough that it tripped the standardization threshold, an imaginary line that when reached almost always ends up motivating some people to work together towards a clarification of the field. In 2018, actors from the industry and academia joined together to form the ZKProof standardization effort, with the goal to "standardize the use of cryptographic zero-knowledge proofs." To this day, it is still an on-going effort, which you can read more about on https://zkproof.org.

You can use general-purpose zero-knowledge proofs in quite a lot of different situations, but to my knowledge they have mostly been used in the cryptocurrency space so far, probably due to the high number of people interested in cryptography and willing to experiment with the bleeding edge stuff. Nonetheless, general-purpose ZKPs have potential applications in a lot of fields: identity management (being able to prove your own age without revealing it), compression (being able to hide most of a computation), confidentiality (being able to hide parts of a protocol), and so on.

The biggest blockers for more applications to adopt general-purpose ZKPs seem to be the following:

1. The large number of ZKP schemes, and the fact that every year more schemes are being proposed.
2. The difficulty of grasping how these systems work and how to use them for specific use cases.

Distinctions between the different proposed schemes are quite important, and a great source of confusion, so here is how some of these schemes are divided:

- **Zero-knowledge, or not**. If some of the information needs to remain secret from some of the participants, then we need zero-knowledgeness. Note that proofs without secrets can be useful as well. For example, you might want to delegate some intensive computation to a service that in turn will have to prove to you that the result they provide is correct.
- **Interactive, or not**. Most ZKP schemes can be made non-interactive (sometimes using the Fiat-Shamir transformation I talked about in chapter 7), and protocol designers seem most interested in the non-interactive version of the scheme. This is because back-and-forths can be time consuming in protocols, but also because interactivity is sometimes not possible. So-called non-interactive proofs are often referred to as **NIZKs** for **non-interactive ZKPs**.
- **Succinct proofs, or not**. Most of the ZKP schemes in the spotlight often refer to themselves as **zk-SNARKs** for **Zero-Knowledge Succinct Non-Interactive Argument of Knowledge**. While the definition may vary, it focuses on the size of the proofs produced by such systems (usually in the order of hundreds of bytes), and the amount of time needed to verify them (within the range of milliseconds). zk-SNARKs are thus short

and easy to verify ZKPs. Note that a scheme not being a zk-SNARK does not disqualify it for the real world; as often different properties might be useful in different use cases.

- **Transparent setup, or not**. Like every cryptographic primitive, ZKPs need a setup to agree on a set of parameters and common values, this is called a **common reference string (CRS)**. But setups for ZKPs can be much more limiting or dangerous than initially thought. There are two types of setup:
  - **Trusted**. This means that whoever created the CRS also has access to secrets that allow them to forge proofs (hence why they are sometimes called "toxic waste"). This is quite an issue, as we are back to having a trusted third party, and schemes that exhibit this property are often the most efficient and have the shortest proof size. That is, the creation of these dangerous parameters can be done in a way that ensures that no single person has access to these secrets.[60]
  - **Transparent**. Fortunately for us, many schemes also offer transparent setups, meaning that no trusted third party need to be present to create the parameters of the system.
- **Universal setup, or not**. There's also a different way to categorize a ZKP's setup, which dictates how flexible the system will be afterwards:
  - **Specific**, which means that the CRS can only be used to prove a single program
  - **Universal**, which means that the CRS can be used for different circuits (potentially of the same size).
- **Quantum-resistant, or not**. Some ZKPs make use of public-key cryptography and advanced primitives like bilinear pairings (which I'll explain later), while some others only rely on symmetric cryptography (like hash functions) which makes them naturally resistant to quantum computers.

Since zk-SNARKs are what's up at the time of this writing, let me give you some intuition as to how they work.

15.3.1 How zk-SNARKs work

First and foremost, there are many many zk-SNARK schemes, too many of them really. But most build on this type of construction:

1. A proving system, allowing a prover to prove something to a verifier.
2. A translation or compilation of a program to something the proving system can prove.

The first part is not too hard to understand, while the second part sorts of requires a graduate course into the subject. So let's take a look at the first part first.

The main idea of zk-SNARKs is that they are all about **proving that you know some polynomial *f(x)* that has some roots**. By roots I mean that the verifier has some values in mind (for example, *1* and *2*) and the prover must prove that the secret polynomial they have in mind evaluates to *0* for these values (for example, $f(1) = f(2) = 0$). By the way, a polynomial that has 1 and 2 as roots (in our example) can be written as $f(x)=(x-1)(x-2)h(x)$ for some polynomial $h(x)$. (If you're not convinced try to evaluate that at $x=1$ and $x=2$.) So we say that the prover must prove that they know an $f(x)$ and $h(x)$ such that $f(x) = t(x)h(x)$ for some target polynomial $t(x) = (x-1)(x-2)$ (in the example that *1* and *2* are the roots that the verifier wants to check).

But that's it, that's what zk-SNARKs proving systems usually provide: something to prove that you know some polynomial. I'm repeating this because the first time I learned about that it made no sense to me: how can you prove that you know some secret input to a program, if all you can prove is that you know a polynomial. Well, that's why the second part of a zk-SNARK is so difficult: it's about translating a program into a polynomial. But more on that later.

Back to our proving system, how does one prove that they know such a function $f(x)$? Well they just have to prove that they know an $h(x)$ such that you can write $f(x)$ as $f(x) = t(x)h(x)$. Ugh… Not so fast here. We're talking about **zero-knowledge** proofs right? How can we prove this without giving out $f(x)$? The answer is in the following three tricks:

1. **Homomorphic commitments.** A commitment scheme similar to the ones we've seen used in other zero-knowledge proofs (covered in chapter 7).
2. **Bilinear pairings**. A mathematical construction that has some interesting properties, more on that later.
3. The fact that **different polynomials evaluate to different values most of the time**.

So let's go through each of them shall we?

15.3.2 Homomorphic commitments to hide parts of the proof

The first trick is to use **commitments** to hide the values that we're sending to the prover. But not only do we hide them, we also want to allow the **verifier** to perform some operations on them so that they can verify the proof. Specifically verify that if the prover commits on their polynomial $f(x)$ as well as $h(x)$, then we have

$com(f(x)) = com(t(x))\ com(h(x)) = com(t(x)h(x))$

where the commitment $com(t(x))$ is computed by the verifier as the agreed constraint on the polynomial. These operations are called **homomorphic operations** and we couldn't have performed them if we had used hash functions as commitment mechanisms (as mentioned in chapter 2). Thanks to these homomorphic commitment, we can "hide values in the exponent" (for example, for a value v then send the commitment $g^v \ mod\ p$) and perform useful computations like equality: if $g^a = g^b$, then $a = b$.

Observe that $g^a = g^b\ g^c$ does not mean that $a = bc$, it means that $a = g^{b+c}$. This is not what we wanted. We can obtain an equation that allows us to check our equality in the exponent like this:

$g^a = (g^b)^c = g^{bc}$

but this works only if c is a known value and not a commitment (for example, g^c). Unfortunately this is a limitation for our proving protocol, as there will be multiplication operations between commitments. Fortunately, cryptography has another tool to get such equations hidden in the exponent: **bilinear pairings**.

15.3.3 Bilinear pairings to improve our homomorphic commitments

This is where bilinear pairings can be used to unblock us, and this is the **sole reason** why we use bilinear pairings in a zk-SNARK (really just to be able to multiply the values inside the commitments). I don't want to go too deep into what bilinear pairings are, but just know that it is just another tool in our toolkit that:

- Takes two values of our group (formed by the values generated by g raised to different powers modulo p) and place them in another group.
- By moving stuff from one group to the other, **we can multiply things that couldn't be multiplied previously**.

So using e as the typical way of writing a bilinear pairing, we have $e(g_1, g_2) = h_3$ where g_1, g_2, and h_3 are generators for different groups. (zk-SNARKs use the same generator on the left: $g_1 = g_2$, which makes the pairing symmetric.) We can use a bilinear pairing to perform multiplications hidden in the exponent via this equation:

$$e(g^b, g^c) = e(g)^{bc}$$

Again, we use bilinear pairings to make our commitments not only homomorphic for the addition, but also for the multiplication. Note that bilinear pairings are used in other places in cryptography, and are slowly making their way as a more common building block: they can be seen in homomorphic encryption schemes (which makes sense after what we've seen here) but also signatures schemes like BLS which I've mentioned in chapter 8.

15.3.4 Where does the succinctness come from? Polynomials

Finally, the **succinctness** of zk-SNARKs comes from the fact that two functions that differ will evaluate to different points most of the time. What this means for us is that if my $f(x)$ is not really equal to $t(x)h(x)$, meaning that I don't have a polynomial $f(x)$ that really has the roots we've chosen with the verifier, then evaluating $f(x)$ and $t(x)h(x)$ at a random point r will not give out the same result **most of the time**. In other words for almost all r, $f(r) \neq t(r)h(r)$. This is known as the **Schwartz-Zippel lemma**, which I pictured in figure 15.12.

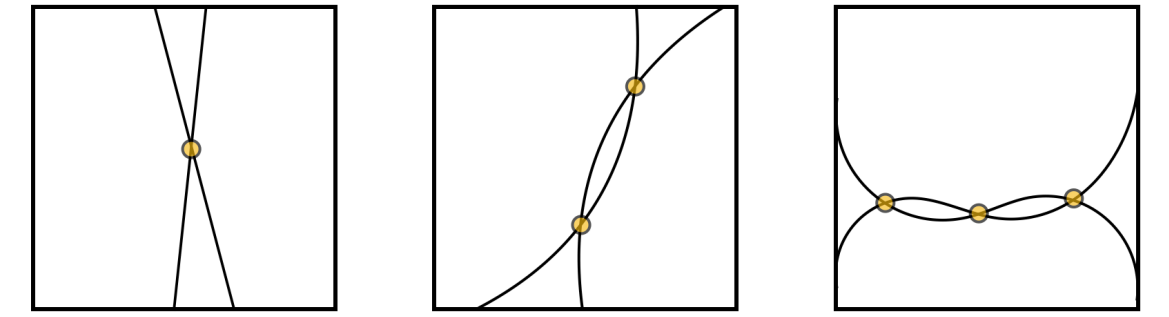


Figure 15.12 The Schwartz-Zippel lemma says that two different polynomials of degree n can intersect in at most n points. In other words, two different polynomials will differ in most points.

Knowing this, it is enough to prove that $com(f(r)) = com(t(r)h(r))$ for some random point r. This is why zk-SNARK proofs are so small: by comparing points in a group you end up comparing much larger polynomials! But this is also the reason behind the "trusted setup" needed in most zk-SNARK constructions. If a prover knows the random point r that will be used to check the equality, then they can forge an invalid polynomial that will still verify the equality. So a trusted setup is about:

1. Creating a random value r.
2. Committing different exponentiation of it g^r, g^{r^2}, g^{r^3}, … so that they can be used by the prover to compute their polynomial without knowing the point r.
3. Destroying the value r.

Does the second point make sense? If my polynomial as the prover is $f(x) = 3x^2 + x$ then all I have to do is compute $(g^{r^2})^3 g^r$ to obtain a commitment of my polynomial evaluated at that random point r (without knowing r).

15.3.5 From programs to polynomials

So far the constraints on the polynomial that the prover must find is that it has some roots (some values that evaluate to 0 with our polynomial). But how do we translate a more general statement into a polynomial knowledge proof? Typical statements in cryptocurrencies (which are the applications currently making the most use of zk-SNARKs these days) are of the form:

- prove that a value is in the range $[0, 2^{64}]$ (this is called a range proof)
- prove that a (secret) value is included in some given (public) Merkle tree
- prove that the sum of some values is equal to the sum of some other values
- and so on…

And herein lies the difficult part. As I said earlier, converting a program execution into the knowledge of a polynomial "sorts of requires a graduate course into the subject." The good news is that I'm not going to tell you all about the details, but tell you enough to give you a sense of

how things work. From there, you should be able to understand what are the parts that are missing from my explanation and fill in the gaps as you wish.

What is going to happen next is:

1. Our program will first get converted into an arithmetic circuit, like the ones we've seen in the section on MPC.
2. That arithmetic circuit will get converted into a system of equations that are of a certain form (called rank-1 constraint system or R1CS).
3. We'll then use a trick to convert our system of equations into a polynomial.

15.3.6 Program are for computers, we need arithmetic circuits instead

First, let's assume that almost any program can be rewritten (more or less easily) in math. The reason why we would want to do that should be obvious: we can't prove code, but we can prove math. For example, let's take the following function where every input is public except for `a` which is our secret input:

```
go
fn my_function(w, a, b) {
  if w == true {
    return a x (b + 3);
   } else {
    return a + b;
   }
}
```

In this simple example, if every input and output is public except for `a`, one can still deduce what `a` is. So this example also serves as an example of what you shouldn't try to prove in zero-knowledge. Anyway, the program can be rewritten in math with this equation:

$w \times (a \times (b + 3)) + (1 - w) \times (a + b) = v$

Where v is the output and w is either 0 (`false`) or 1 (`true`). Notice that this equation is not really a program or a circuit, it just looks like a constraint: if you execute the program correctly, and then fill in the different inputs given and outputs obtained in the equation, the **equality** should be correct. That's one mental step we need to take: instead of executing a function in zero-knowledge (which doesn't mean much really), what we can do is use zk-SNARKs to prove that some given inputs and outputs (secret or public) correctly match the execution of a program.

15.3.7 An arithmetic circuit to an rank-1 constraint system (R1CS)

In any case, we're only one step into the process of converting our execution to something we can prove with zk-SNARKs, the next step is to convert that in a series of constraints that can be converted in the knowledge of a polynomial. What zk-SNARKs want is a **rank-1 constraint system (R1CS)**. R1CS are really just a series of equations that are of the form $L \times R = O$ where L, R, O can only be the addition of some variables, thus the only multiplication is between L and R. It really doesn't matter why we need to transform our arithmetic circuit into such a system of equations, besides that it helps doing the conversion to the final stuff we can prove.

Try to do this with the equation we have, and we can obtain something like

- $a \times (b + 3) = m$
- $w \times (m - a - b) = v - a - b$

We actually forgot the constraint that w is either 0 or 1, which we can add to our system via a clever trick:

- $a \times (b + 3) = m$
- $w \times (m - a - b) = v - a - b$
- $w \times w = w$

Does that make sense? You should really see this system as a set of constraints: if you give me a set of values that you claim match the inputs and outputs of the execution of my program, then I should be able to verify that they also correctly verify these **equalities**. If one of the equalities is wrong, then it must mean that the program does not output the value you gave me for the inputs you gave me.

Another way to think about it is that zk-SNARKs allow you to verifiably remove inputs or outputs of the transcript of the correct execution of a program.

15.3.8 From R1CS to a polynomial

The question is still: how do we transform this system into a polynomial? We're almost there, and as always the answer will be: with a series of tricks!

Since we have three different equations in our system, the first step is to agree on three roots for our polynomial. We can simply choose $1, 2, 3$ as roots, meaning that our polynomial solves $f(x) = 0$ for $x = 1$, $x = 2$, and $x = 3$. Why do that? So that we can make our polynomial represent all the equations in our system at the same time, by representing the first equation when evaluated at 1, and representing the second equation when evaluated at 2, and so on. So the prover's job is now to create a polynomial $f(x)$ such that:

- $f(1) = a \times (b + 3) - m$

- $f(2) = w \times (m - a - b) - (v - a - b)$
- $f(3) = w \times w - w$

And notice that all these equations should evaluate to *0* if the values correctly match an execution of our original program. In other words, our polynomial *f(x)* has roots *1, 2, 3* only if we created it correctly. Now remember, this is what zk-SNARKs are all about: we have the protocol to prove that our polynomial *f(x)* indeed has these roots (known by both the prover and the verifier).

It would be too simple if this is was the end of my explanation, because now the problem is that the prover has way too much freedom in choosing their polynomial *f(x)*, they can simply find a polynomial that has roots *1, 2, 3* without caring about the values *a, b, m, v,* and *w*. They can do pretty much whatever they want. What we want instead, is a system that locks every part of the polynomial except for the secret values that the verifier must not learn about.

15.3.9 It takes two to evaluate a polynomial hiding in the exponent

Let's recap:

- We want a prover that has to correctly execute the program with their secret value *a* and the public values *b* and *w* and obtain the output *v* that they can publish.
- The prover then must create a polynomial by only filling the parts that the verifier must not learn about: the values *a* and *m*.

Thus, in a real zk-SNARK protocol you want the prover to have the least amount of freedom possible when they create their polynomials and then evaluate it to a random point.

To do this, **the polynomial is created somewhat dynamically by having the prover only fill in their part, and having the verifier fill in the other parts**.

For example, let's take the first equation $f(1) = a \times (b + 3) - m$ and represent it as

$$f_1(x) = aL_1(x) \times (b + 3)R_1(x) - mO_1(x)$$

where $L_1(x)$, $R_1(x)$, $O_1(x)$ are polynomials that evaluate to *1* for $x=1$ and to *0* for $x = 2$ and $x = 3$. This is necessary so that they only influence our first equation. (Note that it is easy to find such polynomials via algorithms like Lagrange interpolation.)

Now, notice two things:

- We now have the inputs, intermediate values, and outputs, as coefficients of our polynomials.
- The polynomial *f(x)* is the sum $f_1(x) + f_2(x) + f_3(x)$ where we can define $f_2(x)$ and $f_3(x)$ to represent equations 2 and 3, similarly to $f_1(x)$.

And as you can see, our first equation is still represented at the point $x = 1$:

$$f(1) = f_1(1) + f_2(1) + f_3(1) = f_1(1) = aL_1(1) \times (b + 3)R_1(1) - mO_1(1) = a \times (b + 3) - m$$

With this new way of representing our equations (which remember, represent the execution of our program), the prover can now evaluate parts of the polynomial that are relevant to them by:

1. taking the exponentiation of the random point r hidden in the exponent to reconstruct the polynomials $L_1(r)$ and $O_1(r)$
2. exponentiating $g^{L_1(r)}$ with the secret value a to obtain $(g^{L_1(r)})^a = g^{aL_1(r)}$ which represents $a \times L_1(x)$ evaluated at the unknown and random point $x = r$, and hidden in the exponent
3. exponentiating $g^{O_1(r)}$ with the secret intermediate value m to obtain $(g^{O_1(r)})^m = (g^{mO_1(r)})$ which represents the evaluation of $mO_1(x)$ at the random point r, and hidden in the exponent

And then the verifier can fill in the missing parts by reconstructing $(g^{R_1(r)})^b$ and $(g^{R_1(r)})^3$ for some agreed-on value b using the same techniques the prover used, and then adding the two together to obtain $g^{bR_1(r)} + g^{3R_1(r)}$ which represents the evaluation of $(b + 3) \times R_1(x)$ at the unknown and random point $x = r$, and hidden in the exponent. Finally, the verifier can reconstruct $f_1(r)$ hidden in the exponent by using a bilinear pairing as such:

$$e(g^{aL_1(r)}, g^{(b+3)R_1(r)}) - e(g, g^{mO_1(r)}) = e(g,g)^{aL_1(r) \times (b+3)R_1(r) - mO_1(r)}$$

If you extrapolate these techniques to the whole polynomial $f(x)$ you can figure out the final protocol.

Of course, this is still a gross simplification of a real zk-SNARK protocol, my explanations still leave way too much power to the prover. All the other tricks that are used in zk-SNARKs are meant to further restrict what the prover can do, ensuring that they correctly and consistently fill in the missing parts, as well as optimizing what can be optimized. By the way, the best explanation I've read is the paper "Why and How zk-SNARK Works: Definitive Explanation" by Maksym Petkus, which goes much more in depth and explains all of the parts that I've overlooked.

And that's it for zk-SNARKs. Now this is really just an introduction; in practice, zk-SNARKs are much more complicated to understand, and even use! Not only the amount of work to convert a program into something that can be proven is non-trivial, it sometimes adds new constraints on a cryptography protocol. For example, the mainstream hash functions and signature schemes are often too heavy duty for general-purpose ZKP systems, which has led many protocol designers to investigate different ZKP-friendly schemes.

Furthermore, as I said earlier there are many different zk-SNARKs constructions, and there are also many different non-zk-SNARKs constructions which might be more relevant general-purpose ZKP constructions depending on your use case. But unfortunately, no

one-size-fits-all ZKP scheme seems to exist (for example, a ZKP scheme with a transparent setup, succinct, universal, and quantum-resistant) and it is not clear today which one should use in which cases. The field is still young, and every year new and better schemes are being published. It might be that a few years down the line better standards and easy-to-use libraries will surface. So if you're interested in this space, keep watching!

15.4 Summary

- Many theoretical cryptographic primitives have been making huge progress in terms of efficiency and practicality in the last decade, some of which are making their way in the real world.
- Secure multi-party computation (MPC) is a primitive that allows multiple participants to correctly execute a program together without revealing their respective inputs. Threshold signatures have started to be adopted in cryptocurrencies, while private set intersection (PSI) protocols are being used in modern and large-scale protocols like Google's password checkup.
- Fully homomorphic encryption (FHE) allows one to compute arbitrary functions on encrypted data without decrypting it. It has potential applications in the cloud, where it could prevent access to the data to anyone but the user, while still allowing the cloud platform to perform useful computation on the data for the user.
- General-purpose zero-knowledge proofs (ZKPs) have found many use cases notably with breakthroughs on small proofs that are fast to verify. They have mostly been seen used in cryptocurrencies to add privacy to, or compress the size of, transactions. Their use cases seem broader though, and as better standards and easier-to-use libraries make their way into the real world, we might see them being used more and more.

16 Where cryptography fails and final words

This chapter covers

- The places where issues arise when cryptography is being used.
- The mantras to follow to bake good cryptography.
- The dangers and responsibilities of being a cryptography practitioner.

In this book, you've acquired a sense of the theory, and how it maps to the real world. What's left is for you to actually apply it. To do this, I would expect you to go through a series of steps similar to these ones:

1. You'd find out what are the relevant protocols and/or cryptographic primitives that address your settings or your problem.
2. You'd try to find out if you can use already-existing implementations to implement a solution into your application or system.
3. Perhaps no good implementation already exists, and you'd be confronted with the inevitability of implementing the protocol yourself, hopefully following a specification.

In this chapter, I talk about what can go wrong in any of these steps, as there are a multitude of challenges that someone who seeks to bridge a gap between theory and practice will meet.

16.1 Is this the right protocol? Formal verification to the rescue

If the problem you're facing is a common one to have, chances are that you can simply use a cryptographic primitive or protocol that directly solves your use case. This book gives you a good idea of what the standard primitives and common protocols are, so at this point you should have a good idea of what's at your disposition when faced with a cryptographic problem. Now if you look around you, you'll probably find a bunch of well-respected libraries implementing what you need to use, or maybe even cloud platforms providing easier to use integrations as a service. In any case, I've said it before, I'll say it again, make sure you understand all the fine print of what you're using. As you've seen in this book, misusing cryptographic primitives or protocols can fail in catastrophic ways.

Unfortunately, more often than cryptographers are willing to admit, you will run into trouble when your problem either meets an edge case that the mainstream protocols or libraries don't address, or when your problem doesn't match a standardized solution. For this reason, it is extremely common to see developers creating their own mini-protocols.

This is when trouble starts. When wrong assumptions are made about the primitive's threat model (what it protects against), or about its composability (how it can be used within a protocol), breakage happens. These context-specific issues are amplified by the fact that cryptographic primitives are often built in a silo, where the designer did not necessarily think of all the problems that could arise once the primitive was used in a number of different ways, or within another protocol. I gave many examples of this: X25519 breaking in edge cases protocols (chapter 11), signatures assumed to be unique (chapter 7), ambiguity in who is communicating to whom (chapter 10). So it's not necessarily your fault! The developers have outsmarted the cryptographers, revealing pitfalls that no one knew existed. That's what happened.

If you ever find yourself in this type of situation, **formal verification** can be a wonderful use of your time. Formal verification allows you to write your protocol in some intermediate language, and test some properties on it. For example, the **Tamarin protocol prover** (see figure 16.13) is a formal verification tool that has been (and is) used in order to find subtle attacks in many different protocols.

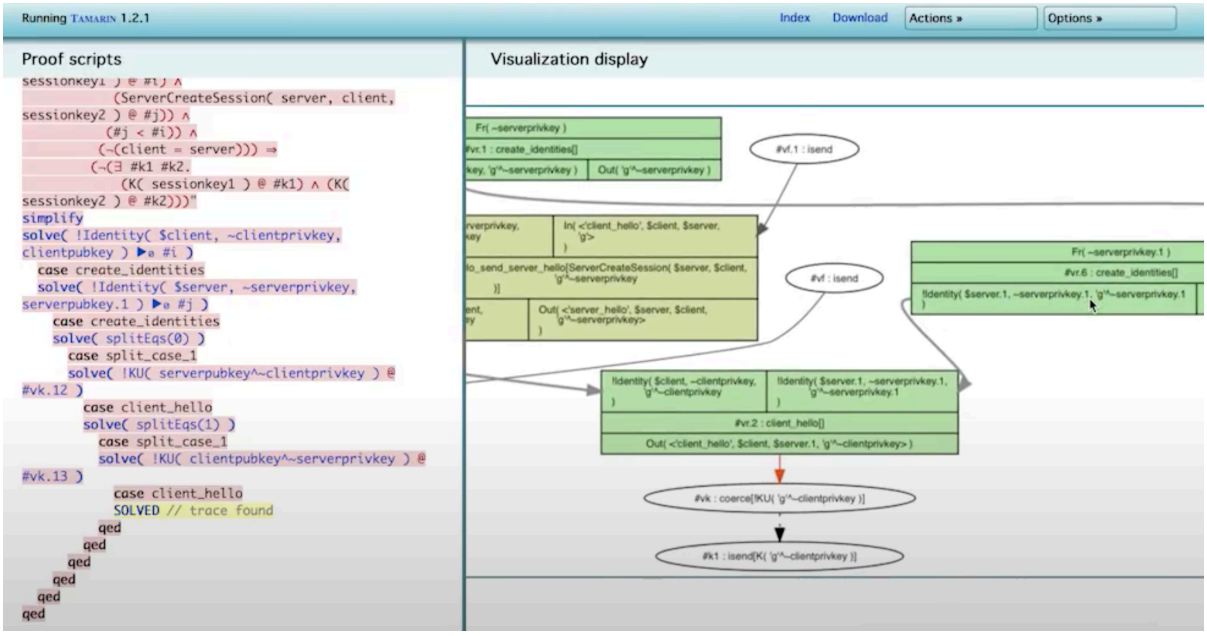


Figure 16.1 The Tamarin protocol prover is a free formal verification tool that can be used to model a cryptographic protocol and find attacks on it.

The other side of the coin is that it is often hard to use formal verification tools. The first step is to understand how to translate a protocol into the language and the concepts used by the tool, which is often not straightforward. After having described a protocol in a formal language, you still need to figure out what you want to prove and how to express it into the formal language as well. It is not uncommon to see a proof that actually proved the wrong things, so one can even ask "who verifies the formal verification?" Some promising research in this area has aimed at making it easier for developers to formally verify their protocols (for example, see the tool Verifpal).

NOTE **It does happen that critical differences are made when writing a formal description of a protocol, compared to the actual protocol being implemented, which then lead to gaps and real-world attacks. This is what happened in 2017, when the KRACK attack (https://krackattacks.com) broke the Wi-Fi protocol WPA2 even though it had been previously formally verified.**

Formal verification is also used to verify cryptographic primitives' security proofs (using formal verification tools like Coq, cryptoverif, and proverif), and even to generate "formally verified" implementations in different languages of cryptographic primitives (see projects like HACL*, Vale, and Fiat-crypto that implement mainstream cryptographic primitives with verified properties like correctness, memory safety, and so on).

That being said, formal verification is not a full proof technique; gaps between the paper protocol and its formal description, or between the formal description and the implementation, will always exist and appear innocuous until found to be fatal.

Bottom line: you need to thoroughly understand what you're using. If you are building a mini-protocol, then you need to be careful and either formally verify that protocol, or ask the community for help.

16.2 You're doing it wrong... About usable security

OK, now let's imagine that you understand that you need to use protocol X for your system. You look around, and you see that there are many libraries or frameworks available for you to use. Which one do you pick? Which is most secure?

It often goes like this:

1. I don't need to figure it out. I host my application in the cloud and they provide a service for that (for example, a key management service backed by secure hardware).
2. I need to figure it out, but the choice is simple: the programming language I use takes care of it (for example, TLS is already implemented in the standard library of Golang).
3. I need to find a library that already implements the protocol or primitive I want to use. Thankfully, I can choose from a number of well-respected libraries (for example, Google's Tink library or the libsodium library are well-known and solid libraries to use in most languages).
4. I need to find a library that already implements the protocol or primitive I want to use, but I'm not sure how secure they are (for example, I found something on Github that seems to do the trick).

No matter what category you're in, you are now a user of cryptography, and unfortunately, most bugs in cryptography happen at this layer: in the usage of cryptography. We've seen that in this book again and again: reusing nonces is bad in algorithms like ECDSA (chapter 7) and AES-GCM (chapter 4), collisions can arise when misuse of hash functions happen (chapter 2), parties can be impersonated due to lack of origin authentication (chapter 9), and so on.

The relevant field of research is called **usable security** and is usually targeted at actual users of applications like secure messaging, but in recent years applied cryptography has shifted to consider developers as users as well. In this sense, cryptographic protocols and libraries have a responsibility to make their interfaces as mis-use resistant as possible.

Treat the developer as the enemy! This was the approach taken by many cryptographic libraries; for example, Google's Tink doesn't let you choose the nonce/IV value in AES-GCM (see chapter 4) in order to avoid accidental nonce reuse; Bernstein's library NaCl chose a fixed set of primitives to support without giving you any freedom in order to avoid complexity; some signing libraries will wrap messages with their signatures, forcing you to verify the signature before releasing the message; and so on.

So far, we're still on the happy path: we're reusing something that is deemed secure, what if we have to implement it ourselves?

16.3 Be boring and polite, a few mentras of cryptography

First and foremost, whatever you're doing, keep it simple. Cryptography is quite an interesting field, and it is going all over the place as new discoveries and primitives are being discovered and proposed, but it is your responsibility to remain conservative. The reason is that **complexity is the enemy of security**. Whenever you do something, it is much easier to do it as simply as possible. This has been dubbed "boring cryptography" by Bernstein in 2015, and has been the inspiration behind the naming of Google's TLS library (BoringSSL).

What if you're in the edge case though? You have to implement a standard yourself, or perhaps you even need to specify a cryptographic protocol. You're now in the realm of what Riad S. Wahby once called **polite cryptography**. Polite cryptography is about cryptographic primitives that leave little room for implementers to hang themselves, and cryptographic specifications that address all edge cases and potential security issues by providing safe and easy interfaces to implement.

In addition, good standards will have accompanying test vectors (inputs that you can feed to your implementation to test its correctness) as well as test frameworks that look for common implementation bugs (for example see Google's Wycheproof which has found a number of bugs in different cryptographic implementations).

Unfortunately, not all standards are "polite," and the cryptographic pitfalls they create are what make most of the vulnerabilities I talk about in this book.

16.4 Cryptography is not an island

Finally, cryptography is not an island. It is often used as part of a more complex system that can also have bugs. Actually, most of the bugs will live in these parts that have nothing to do with the cryptography itself. An attacker often looks for the weakest link in the chain, and it so happens that cryptography often does a good job at raising the bar, whereas encompassing systems which can be much larger and complex can potentially create more accessible attack vectors. Adi Shamir famously said "Cryptography is typically bypassed, not penetrated."

So while it is good to put some effort into making sure that the cryptography in your system is conservative, well-implemented, and well-tested, it is also good to make sure that the same level of scrutiny was applied to the rest of the system. Otherwise, you might have done all of that for nothing.

16.5 Your responsibilities

That’s it, this is the end of the book, you are now free to gallop in the wilderness. But I have to warn you, having read this book gives you no super powers, it should only give you a sense of fragility. A sense that cryptography can easily be misused, and that the simplest mistake can lead to devastating consequences. So proceed with caution. Yet, you now have a big crypto toolset at your belt. You should be able to recognize what type of cryptography is being used around you, perhaps even identify what seems fishy. You should be able to make some design decisions, know how to use cryptography in your application, and understand when you or someone is starting to do something dangerous that might require more attention. Never hesitate to ask for an expert’s point of view.

"**Don’t roll your own crypto**" must be the most overused cryptography line in software engineering. Yet, these folks are somewhat right: while you should feel empowered to implement or even create your own cryptographic primitives and protocols, you should not use it in a production environment. Producing cryptography takes years to get right, years of learning about the ins and outs of the field, not only from a design perspective but from a cryptanalysis perspective as well. Even experts who have studied cryptography all their lives build broken cryptosystems. "Anyone, from the most clueless amateur to the best cryptographer, can create an algorithm that he himself can’t break," Bruce Schneier famously said. At this point, it is up to you to continue studying cryptography. These final pages are not the end of the journey.

Ethics. Finally, I want you to realize that you are in a special position. Cryptography started as a closed field, restricted only to members of the government or academics kept under secrecy, and it slowly became what it is today: a science openly studied throughout the world. But for some people, we are still very much in a time of (cold) war. In 2015 Rogaway drew an interesting comparison between the research fields of cryptography and physics. He pointed out that physics had turned into a highly political field shortly after the nuclear bombing of Japan at the end of World War II. Researchers had started feeling a deep responsibility, as physics was then starting to be clearly and directly correlated to the death of many, and the death of potentially many more. Not much later the Chernobyl disaster would amplify this feeling. On the other hand, cryptography is a field where privacy is often talked as though it is a different subject, and most research is apolitical. Yet, decisions that you and I take can have a long-lasting impact on our society. The next time you design or implement a system using cryptography, think about the threat model you will use. Are you treating yourself as a trusted party, or are you designing things in a way where even you cannot access your users' data or affect their security. How do you empower users through cryptography? What do you encrypt? "We kill people based on metadata," said former NSA Chief Michael Hayden.

In 2012, near the coast of Santa Barbara, hundreds of cryptographers gathered around Jonathan Zittrain in a dark lecture hall to attend his talk "The End of Crypto." This was at Crypto, the most

respected cryptography conference in the world. Jonathan played a clip from the television series *Game of Thrones* to the room. In the video, Varys, an eunuch, poses a riddle to the hand of the king, Tyrion: "Three great men sit in a room: a king, a priest, and a rich man. Between them stands a common sellsword. Each great man bids the sellsword kill the other two. Who lives, who dies?" Tyrion promptly answers, "Depends on the sellsword," to which the eunuch responds, "If it's the swordsman who rules, why do we pretend kings hold all the power?" Jonathan then stopped the clip and pointed to the audience, yelling at them, "You get that you guys are the sellswords, right?"

16.6 Summary

- Real-world cryptography tends to fail mostly in how it is applied. We already know what are good primitives and good protocols to use in most use cases, which leaves their misusage as the source of most bugs.
- A lot of typical use cases are already addressed by cryptographic primitives and protocols. Most of the time, all you'll have to do is find a respected implementation that addresses your problem. Make sure to read the manual, and to understand in what cases you can use a primitive or a protocol.
- Real-world protocols are constructed with cryptographic primitives by combining them like legos. When no well-respected protocols address your problem, you'll have to assemble the pieces yourself. This is extremely dangerous, as cryptographic primitives sometimes break when used in specific situations, or when combined with other primitives or protocols. In these cases, formal verification is an excellent tool to find out issues, although it can be hard to use. Asking the community for help is also a good idea (for example, the "r/crypto" community on reddit, emailing authors directly, or asking the audience at "open mic" sessions at conferences).
- Implementing cryptography is not just hard, you also have to think about hard-to-misuse interfaces, which can be thought of as "usable security" (in the sense that good cryptographic code leaves little room for the user to shoot themselves in the foot).
- Staying conservative and using tried-and-tested cryptography is a good way to avoid issues down the line. Issues stemming from complexity (for example, supporting too many cryptographic algorithms) is a big topic in the community, and steering away from over-engineered systems has been dubbed "boring cryptography." So be as boring as you can.
- Both cryptographic primitives and standards can be responsible for bugs in implementations, due to being too complicated to implement, or being too vague about what implementers should be wary of. Polite cryptography is the idea of a cryptographic primitive or standard that is hard to badly implement. Be polite.
- Cryptography is not an island. If you follow all of the advice this book gives you, chances are that most of your bugs will happen in the non-cryptographic parts of your system. Don't overlook them!
- With what you have learned in this book, make sure to be responsible, and think hard about the consequences of your work.

Notes

1. "TCP/IP Illustrated" is often seen as a reference

2. Specifically, it was the Netscape browser that initated the development of SSL in 1994.

3. Many of the attacks have been summarized in RFC 7457.

4. 2012 - Georgiev et al. - The Most Dangerous Code in the World: Validating SSL Certificates in Non-Browser Software

5. number from netmarketshare.com

6. The most popular way to hide DNS resolution is DNS over HTTPS (DoH).

7. Protocols like Roughtime exist to provide authenticated time to cryptographic applications.

8. Hijacking the Internet Is Far Too Easy https://slate.com/technology/2018/11/bgp-hijacking-russia-china-protocols-redirect-internet-traffic.html

9. RAMPART-A https://en.wikipedia.org/wiki/RAMPART-A

10. in his 1991 essay "Why Do You Need PGP?", Philip Zimmermann ends with "PGP empowers people to take their privacy into their own hands. There’s a growing social need for it. That’s why I wrote it."

11. see the white paper "Efail: Breaking S/MIME and OpenPGP Email Encryption using Exfiltration Channels"

12. github.com/signalapp

13. France enters the Matrix (February 2019) lwn.net/Articles/779331/

14. WhatsApp Encryption Overview whitepaper whatsapp.com/security

15. security.googleblog.com/2017/01/security-through-transparency.html

16. RFC 6819: OAuth 2.0 Threat Model and Security Considerations

17. openid.net

18. bhavukjain.com/blog/2020/05/30/zeroday-signin-with-apple/

19. RFC 2944 - Telnet Authentication: SRP

20. RFC 5054 - Using the Secure Remote Password (SRP) Protocol for TLS Authentication

21. User authentication with passwords, What's SRP? cryptologie.net/article/503/

22. github.com/cfrg/pake-selection

23. this is the case for the IK handshake pattern, for example

24. Sven Laur and Kaisa Nyberg - Efficient Mutual Data Authentication Using Manually Authenticated Strings (2008)

25. Also referred to as *log replication*, *state machine replication*, or *atomic broadcasts*.

26. For a simple and interactive explanation on Raft, check https://thesecretlivesofdata.com/raft

27. To this day, it remains unknown who Satoshi Nakamoto is.

28. https://crypto51.app has a table that lists the cost of performing a 51% attack on different cryptocurrencies based on proof of work.

29. Fischer, Lynch, and Paterson - Impossibility of distributed consensus with one faulty process

30. The DUHK Attack - Don't Use Hard-coded Keys (https://duhkattack.com)

31. In 2017 and 2019, Crypto Experts organized two whitebox cryptography competitions (https://whibox-contest.github.io/2019). Not a single construction managed towithstand the participants' attacks.

32. Europay, Mastercard, and Visa gave their name to the standard, which has been widely adopted in places like Europe while the US has been slow to catch up.

33. Interestingly, in early 2020 Tropic Square (https://tropicsquare.com) was founded with the goal of creating an open-sourced secure element.

34. while YubiKeys are the most popular hardware security tokens, they are not the only ones. SoloKeys, for example, is an interesting open-source alternative (https://solokeys.com).

35. Oswald et al. - Side-Channel Attacks on the YubiKey 2 One-Time Password Generator (2013)

36. TPM Genie: Interposer Attacks Against the Trusted Platform Module Serial Bus (https://nccgroup.com/us/our-research/tpm-genie-interposer-attacks-against-the-trusted-platform-module-serial-bus)

37. The YubiHSM is an HSM that resembles a YubiKey

38. see section 4.11 Mitigation of Other Attacks of FIPS 140-2 (https://csrc.nist.gov/publications/detail/fips/140/2/final)

39. The Untold Story of PKCS#11 HSM Vulnerabilities, 2015 (https://cryptosense.com/blog/the-untold-story-of-pkcs11-hsm-vulnerabilities)

40. Everybody be Cool, This is a Robbery! (2019) (https://donjon.ledger.com/BlackHat2019-presentation)

41. Software Grand Exposure: SGX Cache Attacks Are Practical

42. Foreshadow - Breaking the Virtual Memory Abstraction with Transient Out-of-Order Execution (https://foreshadowattack.eu)

43. SgxPectre Attacks: Stealing Intel Secrets from SGX Enclaves via Speculative Execution

44. RIDL: Rogue In-Flight Data Load (https://mdsattacks.com)

45. Pludervolt (https://plundervolt.com)

46. V0LTpwn: Attacking x86 Processor Integrity from Software

47. SGAxe: How SGX Fails in Practice (https://sgaxe.com)

48. LVI - Hijacking Transient Execution with Load Value Injection (lviattack.eu)

49. Introduction to Trusted Execution Environment: ARM's TrustZone (https://blog.quarkslab.com/introduction-to-trusted-execution-environment-arms-trustzone.html)

50. The Return Of Bleichenbacher's Oracle Threat (ROBOT) Attack (https://robotattack.org)

51. Some algorithms do care about checking if a value is 0 or not in constant time, and there are ways to do this as well. If you're curious to know more about constant-time code, check the Constant-Time Toolkit (https://github.com/pornin/CTTK).

52. Minerva: The curse of ECDSA nonces (minerva.crocs.fi.muni.cz)

53. TPM—Fail TPM meets Timing and Lattice Attacks (tpm.fail)

54. Oscar Reparaz, Josep Balasch and Ingrid Verbauwhede - Dude, is my code constant time? (2016)

55. Neal Koblitz and Alfred J. Menezes wrote "A Riddle Wrapped In An Enigma," a paper speculating on the many possible reasons behind the NSA announcement.

56. Quantum Cryptography: As Awesome As It Is Pointless, published in Wired, 2008

57. RFC 8391 and NIST SP 800-208

58. In his 1982's paper "Protocols for Secure Computations" Andrew C. Yao wrote "Two millionaires wish to know who is richer; however, they do not want to find out inadvertently any additional information about each other's wealth. How can they carry out such a conversation?".

59. Honest-but-curious, also called semi-honest, is a type of participant in MPC protocols that is willing to execute the protocol correctly, but might attempt to learn the other participants' inputs.

60. This is what the Zcash cryptocurrency did with an MPC protocol, involving multiple participants and designed to be secure as long as at least one of them was honest. This meant that one of them had to correctly destroy the secrets they generated during the MPC protocol. This "ceremony" is described in more details in z.cash/technology/paramgen