Chester Rebeiro
Debdeep Mukhopadhyay
Sarani Bhattacharya

Timing Channels in Cryptography

A Micro-Architectural Perspective

Springer

Timing Channels in Cryptography

Chester Rebeiro • Debdeep Mukhopadhyay
Sarani Bhattacharya

Timing Channels in Cryptography

A Micro-Architectural Perspective

Springer

Chester Rebeiro
Columbia University
New York
New York
USA

Sarani Bhattacharya
Department of Computer Science and Engin
IIT Kharagpur
Kharagpur
India

Debdeep Mukhopadhyay
Indian Institute of Technology
Kharagpur
West Bengal
India

ISBN 978-3-319-12369-1 ISBN 978-3-319-12370-7 (eBook)
DOI 10.1007/978-3-319-12370-7

Library of Congress Control Number: 2014954350

Springer Cham Heidelberg New York Dordrecht London

Printed on acid-free paper

Springer is part of Springer Science+Business Media (www.springer.com)

Foreword

With the publication of the Data Encryption Standard DES in 1977 and the invention of public key cryptography in the 1980s, cryptography has moved into civil applications. This was pushed by the electronic revolution, which needs cryptography to create an electronic equivalent of the real world into the digital world. Cryptography is used to provide confidentiality, digital signatures, anonymity, payments, electronic transactions, elections, and many more.

Cryptography and cryptographic protocols by themselves do not provide security: they need a digital platform that executes the algorithms and protocols. In the early days, these platforms were computers and servers sitting in well protected computer rooms and offices. The main concern when implementing the algorithms and protocols was efficiency, as cryptographic algorithms are typically very computationally demanding. The attacker model assumed that the computer platforms themselves were well protected and that only the result of the encryption, the ciphertext, has to travel over insecure communication channels, such as cables or satellite communication or tapes that are transported between bank offices and the bank headquarters.

However, as the electronic revolution expanded, electronics became cheaper, more widespread, and integrated into day to day applications. Examples are pay TV systems, banking cards, identity cards, or access control systems. This omnipresence of electronic gadgets changes the attacker model! Now, the implementation becomes the weak link in the chain: the attacker will attack *while* the electronic device is performing cryptographic operations using the secret key operating on possible sensitive data. During calculations, the devices leak information. These are the so-called side channel attacks. The attacker will observe the device while it performs its calculations. From timing or power consumption variations, or from electromagnetic radiations, the attacker is able to guess which operations and which sensitive data is being handled.

The contributions of this book are situated in this context. One highly important source of side channel information leakage is the timing variations of calculations. Cryptographic algorithms and protocols are written in high level languages and compiled onto sophisticated digital platforms. Both the software stack and the hardware platform include a large variety of optimizations to improve performance. Memory

accesses and branches are expensive in execution time, therefore processors include caches and branch predictors. Unfortunately, these optimizations result in timing variations when executing a program. Caches are considered one of the most important sources of side channel information leakage. This book therefore gives a comprehensive overview and an in depth analysis of many flavors of attacks using caches and branch predictors. It ends with a set of countermeasures to address these timing attacks at different abstraction layers.

This book is a fine read for anyone interested in the many possibilities of cache attacks. A software engineer should not write any line of a cryptographic routine without knowing how powerful timing attacks are. This book gives a clear insight on how these attacks work and how they can be mitigated. Enjoy reading!

Leuven, Belgium Ingrid Verbauwhede
December 2014

Preface

Cryptography plays a vital role in securing e-business and e-commerce transactions. The algorithms used have been rigorously analyzed and tested for their ability to conceal information. However, these algorithms are needed to be realized in systems that are used in a variety of applications. As the saying says, *there's many a slip 'twixt the cup and the lip*, these implementations may leak sensitive information via unintended timing channels. Over the past 20 years, several attacks have been developed that use these *side-channels* to reveal secrets from ciphers. These attacks, known as timing attacks, are powerful enough to break a mathematically robust cipher in few minutes on standard computing platforms.

Starting from its inception to the present day, timing attacks and the underneath statistics have evolved. Enhancements made to computer architecture over the years have also influenced timing channels. To develop a system secure against these threatening channels, one needs to be abreast of the interplay between cryptographic algorithms and computer architecture. Here is where this book steps in. It brings on a single platform aspects of computer architecture and cryptography that are essential to understand *how* timing attacks work and *why* they work. It describes the attacks and analyzes the relationship between the cipher implementation, system micro-architecture, and the attack threat. Various timing channels arising due to cache memories and branch prediction units are presented. The book would help engineers and researchers understand timing attacks and thereafter develop platforms that can tolerate these attacks.
The following topics are covered.

- **Modern Cryptography.** Attackers make use of the cipher's structure to extract the secret key from execution time. The attacks are made even more powerful when ideas from classical cryptanalysis are used with the timing information. The book therefore begins with a review of a variety of modern ciphers ranging from symmetric algorithms like the Advanced Encryption Standard (AES) to asymmetric algorithms like the Rivest-Shamir-Adleman (RSA), and some popular classical cryptanalysis techniques.

- **Superscalar Processor Architectures.** To address how timing channels arise in a system, the book provides a background of the internal micro-architecture of processors. An introduction to modern superscalar architectures has been provided, with an emphasis on the components in them that affect the execution time of ciphers.
- **Time-Driven Cache Attacks on Block Ciphers.** Memory load instructions that cause a cache miss takes considerably longer than those that result in a cache hit. Attackers utilize this difference to build a variety of attacks on block ciphers. A detailed discussion on how these attacks are implemented is presented, with emphasis on various accurate time measurement techniques and analysis methods to observe the cache hit miss phenomenon.
- **Advanced Time-Driven Cache Attacks on Block Ciphers.** Combination of classical cryptanalysis with side-channel attacks paves way to powerful attack methodologies. A combination of classical differential cryptanalysis with cache timing attacks has been discussed in the book to describe this threat.
- **Formal Analysis of Time-Driven Cache Attacks.** Developing suitable metrics for comparison of secured implementations and using them to guide the design flow is an important aspect in security engineering. In this pursuit, the book provides formal modeling of time-driven cache attacks on block ciphers and suggests metrics for evaluation and comparison of such implementations. A discourse on how micro-architectural features, such as pipelining and out-of-order execution, affect cache attacks is presented. These guidelines form a framework for developing ideal implementations of the block ciphers with respect to time-driven cache attacks.
- **Profiled Time-Driven Cache Attacks on Block Ciphers.** There are various types of cache attacks, of which profiled cache attacks are arguably the most powerful. The book provides a detailed description of such profiled attacks, which rely on building timing profiles in a learning phase before mounting the attack. The book also provides insights into developing metrics for estimating the information leakage and relates the leakage to micro-architectural features in modern computers, such as hardware prefetchers.
- **Access-Driven Cache Attacks on Block Ciphers.** Cache attacks come in different flavors, one of them, called access attacks, relies on a spy program running along with the encryption program. The spy uses timing measurements to determine memory access patterns made by the encryption. By monopolizing the OS scheduler, the spy program can accurately determine every memory access made, thereby leading to powerful attacks. The book provides a comprehensive overview on the topic to understand the theory and develop the procedure of such access attacks.
- **Branch Prediction Attacks.** Branch prediction units are artifacts in computer architectures to reduce performance penalties due to branch instructions. They predict whether a branch instruction will be taken or not taken, thereby reducing processor stalls. Athough this can boost performance of an application significantly, it can also lead to timing channels that are catastrophic. Attackers have used these channels to break mathematically strong asymmetric key ciphers such

as the RSA. Several of the attack strategies developed are discussed in detail in the book.

- **Countermeasures**. To circumvent timing attacks, a variety of countermeasures have been proposed at various levels in the system: from the application level and architectural level to the Operating System. The book tries to provide an overview on these countermeasures, how they are applied, their effectiveness, and the consequent overheads involved.

IIT Kharagpur, India
November 2014

Chester Rebeiro
Debdeep Mukhopadhyay
Sarani Bhattacharya

Acknowledgements

This preface would be incomplete without a mention of the people who made this effort possible. The authors would like to express their sincere gratitude to the Computer Science and Engineering Department of IIT Kharagpur, which provides an ideal environment for research on information security. The authors would like to thank all their colleagues, students, collaborators, and funding agencies without whose support this work would not be possible. Special thanks to Abhijit Das, Partha Pratim Chakrabarty, Rajat Subhra Chakraborty, Dipanwita Roy Chowdhury, Phuong Ha Nguyen, and Indranil Sengupta from IIT Kharagpur, and Junko Takahashi (NTT Labs), Toshinori Fukunaga (NTT Labs), and Sobha PM (CDAC Bangalore) for their help and support during the project.

Chester Rebeiro would like to thank his family members especially his parents, wife Sharon, and son Tristan for their love and prayers.

Debdeep Mukhopadhyay would like to thank in particular all his family members and friends. He would like to make a special mention of his parents, wife, and brother for their bonding and wishes. He would like to dedicate this book to his daughter Debanti, who has indeed made his life more interesting.

Sarani Bhattacharya would like to thank her parents and friends for their continuous support and encouragement. She would also like to specially thank her mentor Prof. Debdeep Mukhopadhyay for his invaluable guidance and Dr. Chester Rebeiro for his advice and suggestions.

Contents

List of Abbreviations

AES	Advanced Encryption Standard
BEEA	Binary Extend Euclidean Algorithm
BPU	Branch Prediction Unit
BTB	Branch Target Buffer
CFS	Completely Fair Scheduler
CCA	Chosen Ciphertext Attack
CPA	Chosen Plaintext Attack
CPU	Central Processing Unit
CMOS	Complementary Metal Oxide Semiconductor
CRT	Chinese Remainder Theorem
DOM	Difference of Means
DRAM	Dynamic Random Access Memory
GCD	Greatest Common Divisor
GF	Galois Field
GFN	Generalized Feistel Networks
IPC	Inter Process Communication
ISA	Instruction Set Architecture
L1	Level 1
L2	Level 2
L3	Level 3
MM	Montgomery Multiplication
NIST	National Institute Standards and Technology
NSS	Network Security Services
NTT	Nippon Telegraph and Telephone Corporation
PC	Program Counter
RAM	Random Access Memory
RSA	Rivest Shamir Adleman
SCA	Side Channel Attacks
SMT	Symmetrical Multithreading
SNR	Signal to Noise Ratio
SPN	Substitution Permutation Network
SRAM	Static Random Access Memory
TSC	Time Stamp Counter
VTSC	Virtual Time Stamp Counter

Chapter 1
An Introduction to Timing Attacks

With the ever-increasing proliferation of e-business practices, great volumes of business transactions and data transmission are routinely carried out in devices ranging in scale from personal smartcards to business servers. Cryptographic algorithms are employed in order to secure these transactions and to ensure that the communicated data is only accessible by the intended parties. Exclusive access to the data (referred to as *plaintext*) is achieved by encrypting it with a *key* that is known only to the authorized parties.

Since Claude Shannon's benchmark paper on the communication theory of secrecy systems in [1], significant scientific advancements have been made in the development of cryptographic algorithms. The algorithms in use today are supported by extensive mathematical evaluations, which provide guarantees for secrecy of the plaintext and key. The guarantees for instance, ensure that an attacker would find it extremely difficult to get any information about the secret key from the ciphertext even if the plaintext and cryptographic algorithm are known. To take an example, the best known cryptanalytic attack against the popular Advanced Encryption Standard (AES) cipher would require around 2^{126} operations of the cipher before any information about the key is obtained. This would take several centuries to run even with the best computing resources.

To circumvent the strict security guarantees that cryptographic algorithms comply with, attackers target the physical implementation of the algorithms rather than the algorithms itself. Irrespective of the device that the algorithm is implemented in, its execution would leave a trace in the system and the surrounding medium. The trace may reveal information about the internal state of the cipher that classical cryptanalytic attackers do not have. The mathematical proofs that guarantee security of the cipher fail when such additional information is known. This results in attacks on the cipher that are considerably more practical. For instance, when the cryptographic algorithm executes on a processor, it leaves a trace in the power consumed by the device. This trace, which essentially is a modulation of the dynamic power consumption, contains information about the data being processed and has been used by attackers to retrieve the secret key of ciphers in a few hours. Such channels that leak information about the executing cipher are called *side channels*. The class of attacks that use these side channels to obtain secret information are called

C. Rebeiro et al., *Timing Channels in Cryptography,*
DOI 10.1007/978-3-319-12370-7_1

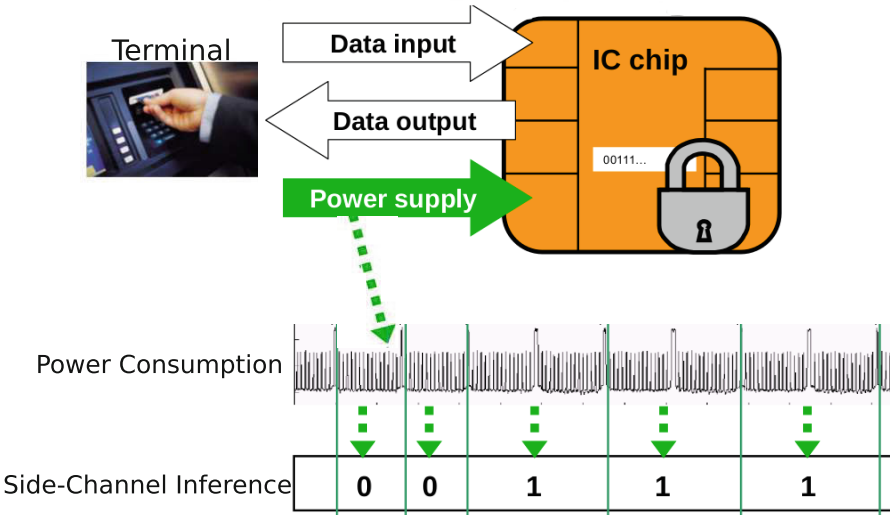


Fig. 1.1 The *power consumption* trace of a smart card reader may reveal the secret. The glitch in the trace can identify a 1 in the secret

side-channel attacks [2]. Figure 1.1 shows how the power consumed by a smart card reader is affected by the secret key stored in the smart card. An attacker monitoring this side can infer the smart card's secret from the glitches in the power trace.

Besides the power consumption, several other side channels have been discovered over the years. For instance, secret keys were recovered from electromagnetic radiation that emanates from a device when executing a cipher [3]. More recently, high-pitched acoustics from the vibration of electronic components have been used to recover cryptographic keys [4]. Side channels based on time have also been used to build attacks against ciphers. Information in timing channels is conveyed by the variation in the execution time of a piece of code [6]. As an example, consider the function Divide in Listing 1.1. The function generally returns the quotient when a is divided by b. However, if b happens to be 0, then the function returns an error value.

Listing 1.1 Function divide returns the quotient of a/b

```
unsigned int Divide (unsigned int a, unsigned int b) {
  if (b== 0)
    return ERROR;
  else
    return a/b;
}
```

Since no division is performed when $b = 0$, the function finishes a bit earlier compared to when $b \neq 0$. An observer who can only monitor the execution time of Divide, would therefore be able to identify invocations when $b = 0$. Further information about b may be obtained from the execution time if the hardware used for

division depends on the divisor length. A division where the divisor is large would complete earlier than a division with a small divisor.

Variation in the execution time of Divide is due to the effect of the input b on the conditional statements and underlying hardware. Conditional statements alter the execution path of the function. This may affect execution time if, for instance, one path is substantially more longer than another. Hardware components in the system could also affect execution time. For example, data present in the cache memory would take considerably less time to load compared to when the data is accessed from the RAM. Other components in the system such as branch prediction units, memory interface units, and ALU blocks may also affect execution time of the program. Microarchitectural features in the processor such as symmetrical multithreading, prefetching, parallelization, pipelining, input–output ports, also influence execution time, albeit in a subtle manner.

To a neophyte it may seem trivial to overcome side-channel attacks by adding noise, randomizing, or fuzzing the side channels of the system. While these additions may successfully stymie certain side-channel attacks, there are still considerable number of attacks that are capable of dealing with such naïve countermeasures. In many of these cases, surprisingly, little extra effort is required to overcome the countermeasures. This book takes a look at how various features in the architecture and program structure are exploited to build time-based side-channel attacks against cryptographic ciphers. It determines how the program design and the choice of architectural components in the system can affect the success of the attacks. In this chapter we provide a brief introduction to side-channel attacks, before introducing timing side channels and the various categories of timing attacks.

1.1 Side-Channel Attacks

The idea of using side channels to steal information has been used for over a century. In World War I, the telephones used in battle fields had just one wire and used the earth to carry the return current. Spies would insert rods in the ground and connect them to amplifiers in order to pick up enemy conversation. During World War II, Bell Labs were the first to discover that electromagnetic emissions from devices could be used for spying. They were able to reveal 75 % of the plaintext that was sent in a secured fashion from a distance of over 80 ft. During the 1950s, the Americans used radiations from encoding machines to spy on encrypted Russian message transmission. By building an appropriate device, it was possible to rebuild messages without decryption. These attacks were studied by American scientists under the code name Tempest, in order to identify shields for equipment. It was initially thought that side-channel attacks would require sophisticated tools that were available only to governments. In 1985, Win van Eck published the first unclassified report which showed how low cost equipment could be used to eavesdrop on messages from a distance of a few hundred meters using the emanations from cathode ray tube monitors. The equipment required an antenna tuned to receive radio transmissions

from the monitor, and could efficiently reconstruct the images that were displayed on the monitor.

More recent studies show how emissions from cables of LCD monitors, wireless keyboards, and LED indicators can be picked up and decoded from several feet away. In the mid-1990s two seminal papers by Paul Kocher showed how execution time and power consumption could be used to easily retrieve secret keys from naïve implementations of cryptographic ciphers. Subsequently, substantial research activities have been directed to understanding side-channel attacks and implementing defenses. The challenge of side-channel attacks comes from the fact that it violates classical notions of cryptography. Side channels expose the fragility of a mathematically sound cipher, making it unsuitable for security applications. Hence a deeper, thorough, and theoretical study of the topic is required. In this section, we provide an overview of the side-channel attacks on cryptographic ciphers.

1.1.1 Side-Channel Attack Requirements

There are three essential requirements in order that a side-channel attack be successfully carried out. These are a perturbation, manifestation, and an observation. We discuss each of these requirements below.

1. **Perturbation :** A perturbation occurs when the secret that the attacker wants to reveal alters the behavior of the system or its state. For instance, in Listing 1.1, the divisor b alters the program execution. Depending on the value of b, the result of conditional statements are altered, thus affecting the instructions that are executed. The behavior of the hardware divider may vary depending on the divisor b. Additionally, hardware registers that hold intermediate results in the divider may be perturbed during the computations. These perturbations in the behavior or system state contain information about the secret divisor b.
2. **Manifestation :** Most often the perturbations in the system, brought about by the execution, cannot be directly accessible by an attacker. In many cases however, they are manifested through side channels, which are generally more accessible. For instance, changes in register values are not always visible to a user executing an application; however, the changes are manifested in the power consumption and electromagnetic radiation of the device. The instructions executed by the program are manifested in the execution time as well as the radiation and power consumption. These manifestations provide information about the program's internals.
3. **Observation :** An attacker would have to observe the side channels in order to obtain the required information about the secret. For example, the power consumption channel would require the attacker to obtain a physical device and monitor its dynamic power consumption through an oscilloscope. For radiation measurements, the attacker would need a receiver and be close enough to pick up emissions from the device. Time-based side channels would require monitoring

of execution time. The clock used to measure time would need to be highly precise in order to distinguish between microarchitectural events in the execution. In certain cases the attacker could create an environment which forces the program or system to behave in a specified way.

1.1.2 The Attacker's Success

Even if the three requirements for a side-channel attack are fulfilled, the attacker may not always be successful in discovering the secret. The amount of success the attacker achieves would depend on additional factors. First, the success would depend on how effectively perturbations convey the secret. For instance, in the function Divide in Listing 1.1, the perturbations due to the conditional statement result in only one value of the divisor to be detected (i.e. when $b = 0$). An attacker would be able to identify cases only when the divisor (b) is zero. All other divisors cannot be distinguished. On the other hand, perturbations in the execution time due to a nonconstant time hardware divider would be able to convey more information about the divisor.

The manifestation of the perturbations through side channels also affects the success of the attack. Perturbations which are well manifested would be more easy to detect. Considering the Divide function again, the presence or absence of the divide operation due to $b = 0$ or $b \neq 0$ has a higher impact on the execution time of the function compared to the subtle variations in time inflicted by the divisor on the divide hardware. As a result, it is easier for an attacker to detect a division compared to detecting the differential behavior of the divider.

A third factor affecting the attack's success is the noise in side channel. Most side-channels are extremely noisy. Further, the amount of noise and their characteristics vary between systems, processors, technologies used, and even depends on the time when measurements are made. An attacker could eliminate a significant amount of noise by signal processing, filtering, and applying other heuristics. Noise, which are uniformly distributed can be eliminated by averaging, provided the attacker is capable of making sufficient number of measurements.

1.1.3 Side-Channel Attack Suppression

Side-channel attacks can be prevented by suppressing or preventing at least one of the afore mentioned requirements. While it is difficult to completely eliminate the perturbations in the system caused when a program executes, the information they convey can be hidden—for instance by masking the secret. Systems could be designed so that manifestations of the perturbations can be eliminated. Take for example, Listing 1.1. The Divide function can be rewritten so that a division is always carried out irrespective of the value of b, for instance, by performing a dummy division when $b = 0$. Further, the division hardware could be redesigned to ensure that the time taken is independent of the value of the divisor b, thereby preventing

any timing channels that occur in the divide operation. Registers in the system can be manufactured in such a way that the power consumed does not depend on the data it stores. As a final resort, the systems could be shielded to ensure that side-channel observations are not made. Electromagnetic shields for instance could be used to prevent radiation-based side-channel attacks. Precision clocks in the system could be disabled to ensure high resolution timers are not available to an attacker, thereby mitigating timing side channels.

Such suppression techniques do not completely protect and in many cases come with large overheads. Consider for instance, huge noise generation circuits added to a mobile phone in order to hide side-channel leakage. This would not only make the phone more bulky but also increase its cost. Worse still would be a large electromagnetic shield around the phone to prevent leakage through radiation. Building efficient and low overhead countermeasures for side-channel attacks is still an open research problem.

1.2 Timing Attacks

Time has always been known to contain information but it was not until 1977 that Schaefer, Gold, Linde, and Scheid published a research paper showing how time could be used to carry information [6]. They showed how a timing channel could be established between two programs in a system to communicate covertly. Information can be transferred from one program to another by the amount of time a sender holds the CPU. For instance, holding the CPU for 10, 20, or 30 μs, can be used to represent a 0, 1, or 2 respectively. Over the years several covert timing channels have been discovered. The rate at which a program performs paging, schedules disks, utilizes cache memory, and scheduled to execute in a CPU have been used to transfer information between two collaborating programs.

In 1996, Kocher published the first timing attack against a cryptographic cipher. He used the variance in the execution time of a cipher to predict secret information. This attack kindled interest in the cryptography community and spurred research in the area, resulting in several papers published on timing attacks and defenses on a variety of ciphers. Timing attacks use the fact that the time required to perform an operation varies depending on the cipher's inputs. The inputs, comprising of a plaintext and key, undergo a series of mathematical operations before the encrypted message is obtained. The mathematical operations performed during encryption affects results of conditional statements, branch destinations, the number and type of instructions executed, instruction operands, and the memory locations accessed by the program. These are the perturbations in the system and could potentially contain information about the secret plaintext and key.

There are two ways by which a timing attacker could monitor these perturbations. The first is by timing executions of the cipher. This works since many of the perturbations in the system made during the execution of the cipher directly affects the execution time. For instance, the pattern with which cache hits and misses occur during execution of the cipher would affect its execution time. Alternatively, measuring the perturbations made by a cipher can be done by another program (called the *spy*).

Algorithm 1.1: Modular Exponentiation

```
   Input: y, x, n
   Output: s = y^x mod n
1  begin
2      s ← 1
3      i ← w − 1
4      while i >= 0 do
5          s ← s^2 mod n
6          if x[i] = 1 then
7              s ← s * y mod n
8          end
9          i ← i − 1
10     end
11     return s
12 end
```

The spy would determine perturbations by monitoring the usage of various hardware components in the system during or soon after the spy executes. This indirect monitoring is generally done with timing channels. For instance, the spy could identify the memory accesses made by the cipher, by using timing to determine how cache memories are utilized.

Just like in any other side-channel attack, a timing attack has two phases: an online phase, followed by an offline phase. The *online phase* monitors the time required for certain operations to be carried out, while the *offline phase* predicts the secret key from the timing information collected. We illustrate how the two phases work by taking a look at Kocher's timing attack.

1.2.1 Kocher's Timing Attack

Paul Kocher's attack was on a naïve implementation of a modular exponentiation algorithm. The algorithm takes as input three numbers y, x, and n, and determines the remainder when y^x is divided by n. This is represented as $y^x \ mod \ n$. The algorithm (depicted in Algorithm 1.1) is an integral part of ciphers such as RSA (Rivest Shamir and Adleman) and DH (Diffie Hellman) based ciphers. Typically, n is public and y can be found by an eavesdropper, while the exponent x is secret. Each iteration of the loop considers a bit of x (denoted $x[i]$ in the algorithm) starting from its most significant bit. Assuming x to be of length w bits, there are w iterations of the loop. The conditional statement in line 6 results in a multiplication being performed only if $x[i] = 1$. Thus, a value of $x[i] = 1$ results in a modular squaring followed by a modular multiplication, while a value of $x[i] = 0$ only has a modular squaring.

The attacker knows the value of n and wants to get x. We assume the attacker is powerful enough to be able to invoke the program (or function) which implements

Algorithm 1.1 and can choose or monitor different values of y. She can also replace x with her own exponent, which we denote x^*.

The attacker triggers the execution of the function with secret x and monitors its execution time. The execution time for the algorithm varies depending on the length of x (*i.e., w*) and its value. This can be denoted as

$$T = e + \sum_{i=0}^{w-1} t_i \ , \tag{1.1}$$

where t_i is the time required for the i-th iteration of the loop (i.e., corresponding to the bit $x[i]$) and e includes the measurement error, loop overhead, and all other sources of inaccuracies. For a fixed x, time distributions can be obtained by observing the execution time corresponding to different values of y. The variance of the distribution is given by

$$Var(T) = Var(e) + w \cdot Var(t). \tag{1.2}$$

The mathematical formalism for the encryption time and its variance is used to recover the secret x. The attack is iterative in the sense that the attacker discovers x starting from its most significant bit. She discovers bit $x[b]$ (where $0 \leq b < w$) only after recovering all bits from $x[w-1]$ to $x[b+1]$.

The attacker makes a guess of $x^*[b]$ (either 0 or 1), chooses $x^* = (x[w-1] \parallel x[w-2] \parallel x[w-3] \parallel \cdots \parallel x[b+1] \parallel x^*[b])$, triggers modular exponentiation, and obtains the execution time (T^*). The difference between the time for the two executions is

$$\begin{aligned} T - T^* &= [e + \sum_{i=0}^{w-1} t_i] - [t_b^* + \sum_{i=b+1}^{w-1} t_i] \\ &= [e + \sum_{i=0}^{b-1} t_i] + t_b - t_b^* \end{aligned} \tag{1.3}$$

If the guess is correct ($x[b] = x^*[b]$), then $t_b = t_b^*$ and the difference between the time for the two executions is $e + \sum_{i=0}^{b-1} t_i$. Further, on a correct guess, the difference in the variance obtained after measuring the execution time of several exponentiation is given by

$$Var[T - T^*] = Var(e) + b \cdot Var(t). \tag{1.4}$$

If the guess is wrong

$$Var[T - T^*] = Var(e) + (b+2) \cdot Var(t). \tag{1.5}$$

In other words, a correct guess causes a reduction in variance, while a wrong guess causes the variance to increase. This distinguisher can be used to validate the guess for $x[b]$.

The accuracy with which a bit in x is determined depends on how a multiplication operation can impact the execution time of a single iteration of the loop. Modern microprocessors have hardware multipliers capable of multiplying two inputs of 32 or 64 bit in a single clock cycle. If the length of s and y are small, a single microprocessor instruction, typically taking less than a nano second would be required to perform the multiplication. In such a case, the impact of the multiplication in the execution time of an iteration is negligible. In spite of this, the multiplication could be detected with reasonable accuracy, provided sufficient amount of time measurements are collected and averaged. As the length of s and y increase beyond the range of the available hardware multiplier, multiple clock cycles would be required to perform the multiplication. Each clock cycle would typically produce a partial product. In this case, the impact of the multiplier in the execution time would increase and the multiplication will be identified with a higher accuracy.

Subsequent to Kocher's work, several timing attacks have been proposed. The attacks target a variety of ciphers, utilize different techniques, and make different assumptions about the attacker. The next part of this section attempts to classify the different timing attacks on ciphers.

1.2.2 Taxonomy of Timing Attacks

Classification Based on the Type of Cipher: Ciphers are broadly classified as symmetric-key and asymmetric-key ciphers. The attack technique varies in each case. Timing attacks on asymmetric-key ciphers are iterative. One bit of the key is recovered at a time just as in Kocher's timing attack on the modular exponentiation algorithm. Timing attacks on symmetric key ciphers target implementations that use look-up tables. Leakage from these ciphers is due to perturbations in the cache memory present in the system.

Classification Based on the Perturbed Component: This classification is based on the component in the system that is perturbed by the cipher's execution. Kocher's timing attack for instance falls into a category based on the execution path, since different execution paths are taken based on the secret. The requirement for these attacks is that the conditional branch depends on the value of the cipher's secret. Further, the paths should have different execution time requirements. These attacks are generally relevant to public-key ciphers.

A closely related category of timing attacks target perturbations in the processor's branch predictor. These predictors are present in most modern processors in order to reduce the performance overheads when a branch occurs. These units automatically predict the branches during the execution of the program and fetch instructions from the new location. Branch predictors can substantially influence execution time and have been exploited by attackers to determine the secret from public key ciphers.

Perturbations made to cache memories have been used by attackers to discover secret keys from ciphers, especially block ciphers. Attacks that use cache memories require that the cipher accesses memory at locations which depend on the secret key.

Timing attacks based on cache memories use the fact that a cache miss takes considerably longer time compared to a cache hit. This is manifested in the execution time.

Classification Based on the Attacker's Capabilities: Attackers capabilities may differ. The most powerful attacker is one who can choose the plaintext to be encrypted, trigger encryptions, accurately monitor the side channel after every instruction executed. On the other side of the spectrum, the weakest attacker is one who can only monitor execution time remotely, for instance over a network. The capabilities of the attacker would therefore dictate the attack's success.

1.3 Organization

A primary goal of this book is to introduce and analyze timing attacks on block ciphers due to information leaking from cache memories in the system. The book discusses how microarchitectural features in the cache memory such as automatic hardware prefetching, nonblocking, parallel, and pipelined servicing of cache misses can influence leakage. Mathematical analysis is used to quantify this leakage. Another goal of the book is to introduce and analyze timing attacks on asymmetric key ciphers due to information leakage in the processor's branch prediction units. The instruction flow of the asymmetric ciphers can be revealed using the information leakage from the branch predictors. In this book, using timing as side channel, we discuss how the microarchitectural features can be monitored efficiently, and due to their deterministic nature they eventually result in leaking the secret instruction flow. The organization is as follows.

In Chapter 2, a brief review of modern cryptography is given. Topics addressed include an overview of encryption algorithms, their implementation, and classical cryptanalysis. Covert timing channels used for stealthy communication is introduced, and a brief introduction to formal analysis of side-channel attacks is presented.

The aim of Chapter 3 is to provide an overview of superscalar processor architecture, modern cache memories, and branch predictors used in modern computer systems. Subtle changes in these components can significantly affect information leakage from timing channels.

Chapter 4 discusses how time measurements can be accurately made on systems. Accurate time measurements are critical for the success of most timing attacks. The chapter then discusses how cache hits and misses can be distinguished from the execution time. These hits and misses are used to build a time-driven attack on a block cipher.

The information obtained from the attack presented in Chapter 4 is restricted by the size of the cache line. Chapter 5 shows how this restriction can be overcome by advanced attack strategies. The chapter shows how block cipher algorithms and their differential properties can be exploited to build powerful time-driven cache attacks.

Chapter 6 develops a framework to analyze information leakage in time-driven cache attacks. The framework is capable of analyzing modern cache memories that

are equipped with microarchitectural acceleration features such as nonblocking, parallel, pipelined, and out-of-order servicing of cache misses. The framework is used to evaluate the leakage of popular block implementations and also identify the ideal way a block cipher should be implemented.

Chapter 7 introduces profiled time-driven cache attacks. These attacks require a powerful adversary who can characterize the system behavior when the cipher is executed. The adversary's success would differ depending on the way the block cipher is implemented. Further, automatic hardware prefetching can abet the attack.

Chapter 8 provides an illustrative description of access-driven attacks on block ciphers. Just as in the attacks described in Chap. 4, these attacks track cache trace patterns made by an executing block cipher. The tracing is done by a malicious process which schedules itself in such a manner, so as to observe the cache traces of individual accesses made by the cipher. The chapter explains various strategies used by the malicious process in order to make fine-grained timing measurements to observe several microarchitectural events.

Chapter 9 starts with an introduction to the algorithms used in the implementation of the public key cipher RSA. The branch predictors play a major role in predicting results of branch conditions in the cipher. In this chapter we provide several implementations where the leakage in the conditional instructions present in RSA are exploited using time as a side channel.

In Chap. 10, we highlight some of the important countermeasures that have been proposed for time-driven attacks. These countermeasures differ in how they eliminate leakage, the location in the system that they are applied, and the performance overhead.

Reference

1. Shannon CE (1949) Communication theory of secrecy systems. Bell Syst Tech J 28, 656–715
2. Kocher PC, Jaffe J, Jun B (1999) Differential power analysis, CRYPTO'99 Proceedings of the 19th Annual International Cryptology Conference on Advances in Cryptology, pp 388–397. Springer-Verlag London
3. Agrawal D, Archambeault B, Rao JR, Rohatgi P (2003) The EM Side-Channel(s), Cryptographic Hardware and Embedded Systems – CHES 2002, pp 29–45, Springer, Berlin
4. Genkin D, Shamir A, Tromer E (2013) RSA Key Extraction via Low-Bandwidth Acoustic Cryptanalysis, Cryptology ePring Archive, Report 2013/857
5. Schaefer M, Gold B, Linde R, Scheid J (n.d.) Program Confinement in KVM/370, ACM'77 Proceedings of the 1977 annual conference, pp 404–410, ACM New York, NY, USA
6. Kocher P (1996) Timing Attacks on Implementations of Diffie-Hellman, RSA, DSS, and other Systems. CRYPTO'96 Proceedings of the 16th Annual International Cryptology Conference on Advances in Cryptology, pp 104–113, Springer-Verlag, London, UK

Chapter 2
Modern Cryptography

The art of cryptographic algorithms is an ever evolving field. Initiating from prehistoric times, the main objective of cryptographic algorithms have been to protect and allow usage of information in a legal manner. Encryption is a process of converting a plaintext message into a piece of random-looking text, often called ciphertext. The ciphertext, ideally, should contain or transfer no information to the curious third party, often referred to as the adversary. In order to allow the intended receiver to obtain back the message from the ciphertext, every encryption algorithm is reversible. Thus, the inverse operation of obtaining back the plaintext from the ciphertext, is called decryption. According to the wisdom in cryptography, the algorithms for encryption and decryption are always published and known to even the adversary. Then what does the security rely on? The mappings (from plaintext to ciphertext and vice versa) depends on an information called the key, which is hidden from the attacker. While the goal of cryptography is to design and construct such ciphering algorithms, the objective of cryptanalysis is to develop techniques to obtain the key more efficiently than making a random guess on the key. There are different classes of cryptographic algorithms, depending on their objectives. While secrecy is the obvious requirement, there are other important goals too; for example, algorithms to guarantee the integrity of information, or methods to ensure that one cannot deny a commitment to a transaction (often called the property of nonrepudiation). Then there are algorithms which ensures that authenticity is maintained in a communication, and legal parties can trust with whom they are communicating. In this book we mainly target cryptographic algorithms, with respect to secrecy of data. But often these constructions can be used for achieving the other requirements, namely authentication and nonrepudiation. Hash functions, which are used for integrity checks are not in the scope of this book.

2.1 Types of Encryption Algorithms

For ciphers, there is an encryption key (K_a) and a decryption key (K_b), which are equal for certain classes of algorithms (called symmetric-key ciphers) and different for the other class (called asymmetric-key ciphers). The scenario of a cryptographic

C. Rebeiro et al., *Timing Channels in Cryptography,*
DOI 10.1007/978-3-319-12370-7_2

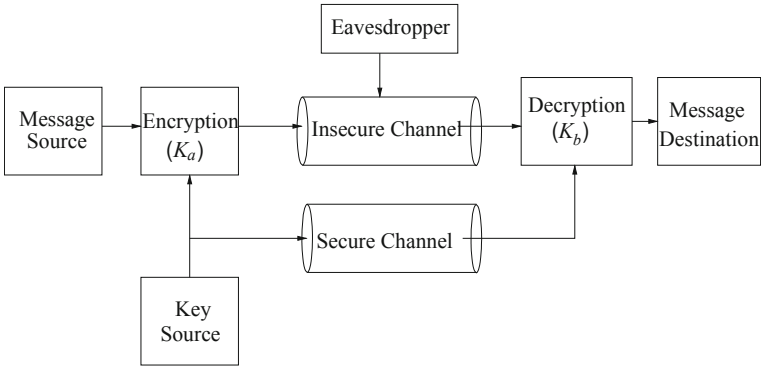


Fig. 2.1 Secret key cryptosystem model

communication is illustrated in Fig. 2.1. The encryptor uses a key K_a and the decryptor a key K_b, where depending on the equality of K_a and K_b there are two important classes of cryptographic algorithms. More precisely, the two classes of ciphers are:

- Private-key (or symmetric) ciphers: These ciphers have the same key shared between the sender and the receiver. Thus, referring to Fig. 2.1 $K_a = K_b$.
- Public-key (or asymmetric) ciphers: In these ciphers we have $K_a \neq K_b$. The encryption key and the decryption keys are different.

As in symmetric-key or private-key algorithms both the encryptor and decryptor use the same key, it must somehow be securely exchanged before secret key communication can begin. The key exchange is a major bottleneck and for n-parties in a network, the number of key exchanges required can grow quite fast (${}^nC_2$ ways). However, these algorithms are often fast and are used for bulk data encryption. Two important subclasses of symmetric-key algorithms are block ciphers and stream ciphers. Block ciphers, as the name suggest operates on fixed blocks or chunks of data, while stream ciphers operate on bits or few bits of data (thus the encryption takes place like a stream!). The Advanced Encryption Standard (AES) is a very popular block cipher, while Trivium is a popular stream cipher.

Public-key algorithms, on the other hand, provide a nice solution to the key-exchange problem. In such algorithms, as we discussed the encryption and decryption keys are different. The algorithms have a key pair, consisting of (i) Public key, which can be freely distributed and is used to encrypt messages. In Fig. 2.1, this is denoted by the key K_a, and (ii) Private key, which must be kept secret and is used to decrypt messages. The decryption key is denoted by K_b in the Fig. 2.1.

In the public key or asymmetric ciphers, the two parties—often called Alice and Bob—are communicating with each other and have their own key pair. They distribute their public keys freely. Mallory (or the adversary) has the knowledge of not only the encryption function, the decryption function, and the ciphertext, but also has the capability to encrypt the messages using Bob's public key. However, she is unaware of the secret decryption key, which is the private key of the algorithm. The security

of these classes of algorithms rely on the assumption that it is computationally hard or complex to obtain the private key from the public information. Doing so would imply that the adversary solves a mathematical problem which is widely believed to be difficult. It may be noted that we do not have any proofs for their hardness; however, we are unaware of any efficient techniques to solve them. The elegance of constructing these ciphers lies in the fact that the public keys and private keys still have to be related in the sense, that they perform the invertible operations to obtain the message back. This is achieved through a class of magical functions, which are called *one-way* functions. These functions are easy to compute in one direction, while computing the inverse from the output is believed to be a difficult problem. RSA is a famous public-key algorithm for this class of ciphers.

Example 2.1 This cipher is called the famous RSA algorithm (Rivest Shamir Adleman). Let $n = pq$, where p and q are properly chosen and large prime numbers. Here the proper choice of p and q are to ensure that factorization of n is mathematically complex. The plaintexts and ciphertexts are $P = C = Z_n$, the keys are $K_a = \{n, a\}$ and $K_b = \{b, p, q\}$, such that $ab \equiv 1 \bmod \phi(n)$. The encryption and decryption functions are defined as, $\forall x \in P$, $e_{K_a}(x) = y = x^a \bmod n$ and $d_{K_b}(y) = y^b \bmod n$.

Both symmetric-key ciphers and asymmetric-key ciphers are widely studied. We provide a quick overview on some facts, which we use in the book.

2.2 Block Ciphers: An Important Family of Symmetric-Key Ciphers

The book develops a general theory for time-driven cache attacks on block ciphers. The theory can be applied to any cipher with an iterative structure that is implemented with look-up tables. To test the theory we selected few ciphers, like AES, CAMELLIA, and CLEFIA. AES was chosen because it is the world wide standard. It has a substitution permutation network (SPN) structure and the implementation generally used has 5 large tables of 1024 bytes. The first four tables are invoked 36 times per 128-bit encryption, while the last table is invoked 16 times. Many of the latest CPUs support dedicated instructions for AES [1], on which cache attacks fail. Besides SPN, most cipher designs follow the Feistel structure (Fig. 2.2). The figure shows the rth round of a Feistel block cipher. The block is divided into two parts, L_r (Left) and R_r (Right), which are recursively computed as $L_r = R_{r-1} \oplus \mathbf{F}(L_{r-1}, k_r)$, and $R_r = L_{r-1}$. The transformation $\mathbf{F}$ is composed of several nonlinear transformations or S-Boxes, and combine the round key k_r with a portion of the block. The reversibility of the round does not depend on whether the nonlinear layer $\mathbf{F}$ is invertible or not. The rounds are repeated for certain number of iterations to ensure sufficient security margin against known attacks.

We chose CAMELLIA as a representative of these ciphers. The third cipher we chose is CLEFIA, which has a generalized Feistel structure [2] and a round function which is conceptually similar to that of AES. Further, unlike the AES implementation considered, CLEFIA and CAMELLIA implementations used small tables of 256 bytes. This section provides a brief description of each cipher algorithm.

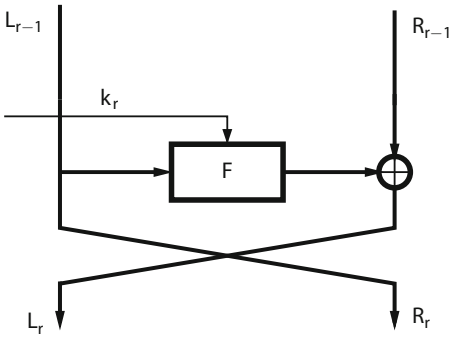


Fig. 2.2 Feistel structure of a block cipher

2.2.1 AES

In 2001, the National Institute of Standards and Technology (NIST) recommended the use of Rijndael as the AES [3]. AES is a symmetric-key block cipher and can use key sizes of either 128, 192, or 256 bits to encrypt and decrypt 128-bit blocks. We summarize the AES-128 standard, which uses a key size of 128 bits. The input to AES-128 is arranged in a 4×4 matrix of bytes called *state*. The state undergoes a series of transformations in ten rounds during the encryption process.

Algorithm 2.1 presents the AES-128 algorithm. The first operation on the input is the AddRoundKeys, which serves to provide the initial randomness by mixing the input key. The state is then subjected to nine rounds to further increase the diffusion and confusion in the cipher [4]. Each round comprises four operations on the state: SubBytes, ShiftRows, MixColumns, and AddRoundKeys. The state is then subjected to a final round, which has all operations except the MixColumns operation.

The four AES operations are defined as follows:

Algorithm 2.1: AES-128

Input: 128 bit plaintext input block $\mathbf{x}$ and 128 bit round keys $\mathbf{k^{(0)}}, \mathbf{k^{(2)}}, \cdots, \mathbf{k^{(10)}}$ generated from a secret key using the AES-128 key generation algorithm
Output: 128 bit ciphertext

```
1  begin
2      r ← 1
3      s^(0) ← AddRoundKeys(x, k^(0))
4      while r <= 9 do
5          s^(r) ← SubBytes(s^(r−1))
6          s^(r) ← ShiftRows(s^(r))
7          s^(r) ← MixColumns(s^(r))
8          s^(r) ← AddRoundKeys(s^(r), k^(r))
9      end
10     s^(10) ← SubBytes(s^(9))
11     s^(10) ← ShiftRows(s^(10))
12     y ← AddRoundKeys(s^(10), k^(10))
13     return y
14 end
```

- AddRoundKeys : Each element in the state is subjected to a bitwise ex-or with a 128-bit round key. The round key is generated from the secret key by a key expansion algorithm as in Algorithm 2.2 [4].
- SubBytes : Each element in the state is replaced by an affine transformation of its inverse in the field $GF(2^8)$. For a byte s_i in the state, this operation is denoted by $S(s_i)$.
- ShiftRows : Provides a cyclic shift of the ith row in the state by i bytes toward the left (where $0 \leq i \leq 3$). That is, each byte in the ith row is cyclically shifted to the left by i bytes.
- MixColumns : Provides a column-wise linear transformation of the state matrix. Each column of the state matrix is considered as a polynomial of degree 3 with coefficients in $GF(2^8)$ and multiplied by the polynomial $\{03\}\alpha^3 + \{01\}\alpha^2 + \{01\}\alpha + \{02\}$ mod $(\alpha^4 + 1)$. The combination of ShiftRows and MixColumns provide the necessary diffusion for the cipher.

Key Expansion Algorithm [3] takes the secret key as input and generates round keys for 11 AddRoundKeys operations performed in AES. Key expansion as in Algorithm 2.2 uses two operations ROTWORD and SUBWORD which performs cyclic shift and substituition of four bytes (B_0, B_1, B_2, B_3) as

$$ROTWORD(B_0, B_1, B_2, B_3) = (B_1, B_2, B_3, B_0) \tag{2.1}$$

and

$$SUBWORD(B_0, B_1, B_2, B_3) = (SubBytes(B_0), SubBytes(B_1), SubBytes(B_2), SubBytes(B_3)) \tag{2.2}$$

Algorithm 2.2: Key Expansion of AES-128

Input: 128 bit secret key
Output: 11 round keys each of 4 words as $w[0], \cdots, w[43]$

begin
  $RCon[1] \leftarrow 01000000$
  $RCon[2] \leftarrow 02000000$
  $RCon[3] \leftarrow 04000000$
  $RCon[4] \leftarrow 08000000$
  $RCon[5] \leftarrow 10000000$
  $RCon[6] \leftarrow 20000000$
  $RCon[7] \leftarrow 40000000$
  $RCon[8] \leftarrow 80000000$
  $RCon[9] \leftarrow 1B000000$
  $RCon[10] \leftarrow 36000000$
  for $i <= 0$ *to* 3 **do**
    $w[i] \leftarrow (k[4i], k[4i+1], k[4i+2], k[4i+3])$
  end
  for $i <= 4$ *to* 43 **do**
    $temp \leftarrow w[i-1]$
    if $i \equiv 0(\ mod\ 4)$ **then**
      $temp = SUBWORD(ROTWORD(temp)) \oplus RCon[i/4]$
    end
    $w[i] \leftarrow w[i-4] \oplus temp$
  end
  return $(w[0], \cdots, w[43])$
end

In addition to this, AES involves a round constant term that is defined as $RCon[1], \cdots, RCon[10]$ as constants in hexadecimal before the key expansion Algorithm 2.2.

Starting from the 4×4 byte state, Fig. 2.3 shows the transformation it undergoes in a round (for $1 \leq r \leq 9$).

2.2.1.1 Software Implementations of AES

Of all operations, the SubBytes is the most difficult to implement. On 8-bit microcontrollers, a 256-byte look-up table is ideal to perform this operation. The table provides the necessary flexibility in terms of content, small footprint, and speed. For 32-bit platforms, more efficient implementations can be built using larger tables. We give a brief description of this method, which is known as T-table implementations. T-table implementations were first proposed in [5] and has been adopted by several crypto-libraries such as OpenSSL[1].

Consider four look-up tables defined as follows:

$$\mathcal{T}_0[z] = \begin{bmatrix} 02 \bullet S(z) \\ S(z) \\ S(z) \\ 03 \bullet S(z) \end{bmatrix}; \mathcal{T}_1[z] = \begin{bmatrix} 03 \bullet S(z) \\ 02 \bullet S(z) \\ S(z) \\ S(z) \end{bmatrix}; \mathcal{T}_2[z] = \begin{bmatrix} S(z) \\ 03 \bullet S(z) \\ 02 \bullet S(z) \\ S(z) \end{bmatrix};$$
$$\mathcal{T}_3[z] = \begin{bmatrix} S(z) \\ S(z) \\ 03 \bullet S(z) \\ 02 \bullet S(z) \end{bmatrix} \tag{2.3}$$

Each table is of 1024 bytes mapping a byte z of the state to a 32-bit value. Using these tables, the first nine AES rounds can be expressed as follows

$$\begin{aligned} \mathbf{s}^{(r+1)} = & \mathcal{T}_0[s_0^{(r)}] \oplus \mathcal{T}_1[s_5^{(r)}] \oplus \mathcal{T}_2[s_{10}^{(r)}] \oplus \mathcal{T}_3[s_{15}^{(r)}] \oplus [k_0^{(r)}\ k_1^{(r)}\ k_2^{(r)}\ k_3^{(r)}]^T \parallel \\ & \mathcal{T}_0[s_4^{(r)}] \oplus \mathcal{T}_1[s_9^{(r)}] \oplus \mathcal{T}_2[s_{14}^{(r)}] \oplus \mathcal{T}_3[s_3^{(r)}] \oplus [k_4^{(r)}\ k_5^{(r)}\ k_6^{(r)}\ k_7^{(r)}]^T \parallel \\ & \mathcal{T}_0[s_8^{(r)}] \oplus \mathcal{T}_1[s_{13}^{(r)}] \oplus \mathcal{T}_2[s_2^{(r)}] \oplus \mathcal{T}_3[s_7^{(r)}] \oplus [k_8^{(r)}\ k_9^{(r)}\ k_{10}^{(r)}\ k_{11}^{(r)}]^T \parallel \\ & \mathcal{T}_0[s_{12}^{(r)}] \oplus \mathcal{T}_1[s_1^{(r)}] \oplus \mathcal{T}_2[s_6^{(r)}] \oplus \mathcal{T}_3[s_{11}^{(r)}] \oplus [k_{12}^{(r)}\ k_{13}^{(r)}\ k_{14}^{(r)}\ k_{15}^{(r)}]^T \end{aligned} \tag{2.4}$$

A byte of the state in the current round ($\mathbf{s}^{(r)}$) is denoted $s_i^{(r)}$ and the next round state is denoted $\mathbf{s}^{(r+1)}$, where $0 \leq r \leq 9$ and $0 \leq i \leq 15$. The final round cannot use these tables due to the absence of the MixColumns operation.

[1] http://www.openssl.org.

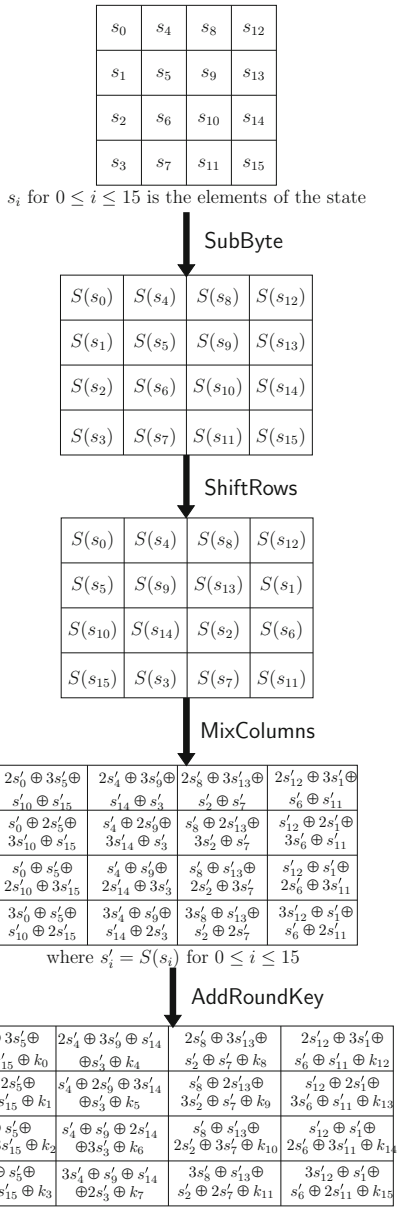


Fig. 2.3 Transformations of the state in a round of AES ($1 \leq r \leq 9$)

2.2.2 CLEFIA

CLEFIA is a 128-bit block cipher developed by Sony [6] and is currently incorporated in ISO/IEC 29192-2 as a light-weight block cipher standard. The specification [6, 7] defines three key lengths of 128, 192, and 256 bits. This book considers 128-bit keys though the results are valid for the other key sizes also. The structure of CLEFIA is shown in Fig. 2.4. The input has 16 bytes, P_0 to P_{15}, grouped into four byte words. There are 18 rounds, and in each round, the first and third words are fed into nonlinear functions $F0$ and $F1$ respectively. The output of $F0$ and $F1$ are ex-ored with the second and fourth words. Additionally, the second and fourth words are also whitened at the beginning and end of the encryption. The F functions take four input bytes and four round keys. The nonlinearity in the F functions are due to two 256 element s-boxes $S0$ and $S1$. Matrices $M0$ and $M1$ diffuse the outputs of the s-boxes. They are defined as follows:

$$M0 = \begin{pmatrix} 1 & 2 & 4 & 6 \\ 2 & 1 & 6 & 4 \\ 4 & 6 & 1 & 2 \\ 6 & 4 & 2 & 1 \end{pmatrix} \qquad M1 = \begin{pmatrix} 1 & 8 & 2 & A \\ 8 & 1 & A & 2 \\ 2 & A & 1 & 8 \\ A & 2 & 8 & 1 \end{pmatrix} \tag{2.5}$$

The design of the s-boxes $S0$ and $S1$ differs. $S0$ is composed of four s-boxes $SS0$, $SS1$, $SS2$, and $SS3$; each of 16 bytes. The output of $S0$ is given by :

$$\begin{aligned} \beta_l &= SS2[SS0[\alpha_l] \oplus 2 \cdot SS1[\alpha_h]] \\ \beta_h &= SS3[SS1[\alpha_h] \oplus 2 \cdot SS0[\alpha_l]] \end{aligned} \tag{2.6}$$

where $\beta = (\beta_h|\beta_l)$, $\alpha = (\alpha_h|\alpha_l)$, and $\beta = S0[\alpha]$. The output of $S1$ for the input byte α is given by $g((f(\alpha))^{-1})$, where g and f are affine transforms and the inverse is found in the field $GF(2^8)$.

The CLEFIA encryption algorithm has four whitening keys WK_0, WK_1, WK_2, and WK_3; and 36 round keys $RK_0, \ldots, RK_{35}$. Key expansion is a two-step process. First, a 128-bit intermediate key L is generated from the secret key K using a GFN function [7]. From this, the round keys and whitening keys are generated as shown below:

```
Step 1: WK0|WK1|WK2|WK3 ← K
Step 2: For i ← 0 to 8
            T ← L ⊕ (CON24+4i|CON24+4i+1|CON24+4i+2|CON24+4i+3)
            L ← Σ(L)
            if i is odd: T ← T ⊕ K
            RK4i|RK4i + 1|RK4i + 2|RK4i + 3 ← T
```

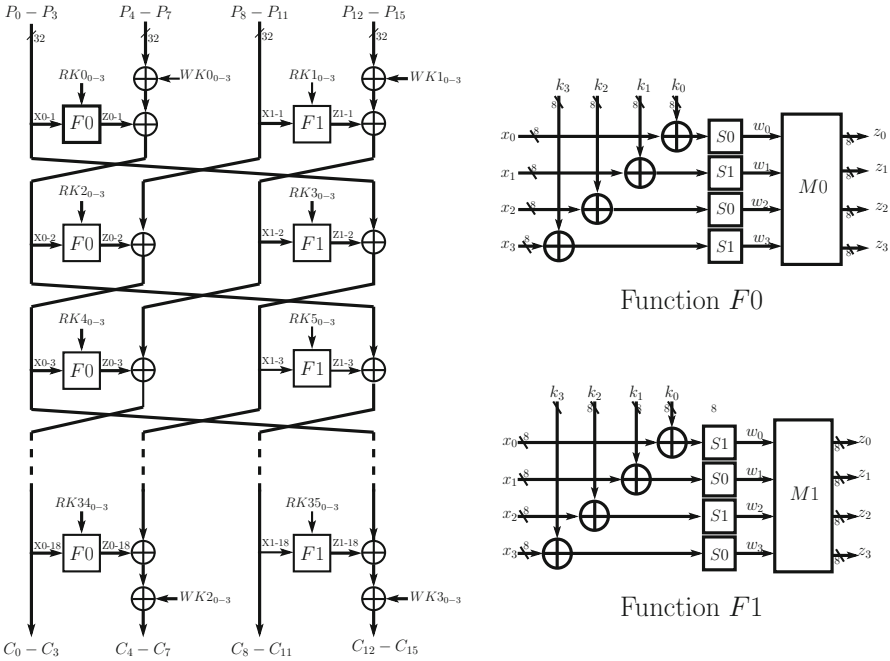
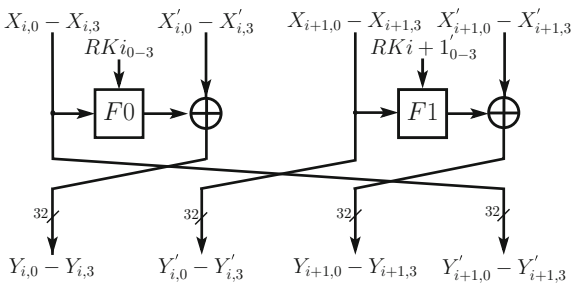


Fig. 2.4 CLEFIA block diagram

Fig. 2.5 A round of CLEFIA



The function Σ, known as the *double swap* function, rearranges the bits of L.

$$\Sigma(L) \leftarrow L_{(7\cdots63)}|L_{(121\cdots127)}|L_{(0\cdots6)}|L_{(64\cdots120)} \tag{2.7}$$

In the later part, thirty-six 32-bit constant values CON_i $(24 \leq i < 60)$ are used.

Just like AES, CLEFIA can be implemented with T-tables. The T-table implementation for CLEFIA is in the Sect. 2.2.2.1.

2.2.2.1 T-Table Implementation of CLEFIA

A round of CLEFIA has 128-bit input and produces 128-bit output. Each input and output is grouped as four words as shown in Fig. 2.5. The function $F0$ is defined as

follows,

$$F0: \begin{bmatrix} Z_{i,0} \\ Z_{i,1} \\ Z_{i,2} \\ Z_{i,3} \end{bmatrix} = \begin{bmatrix} 1 & 2 & 4 & 6 \\ 2 & 1 & 6 & 4 \\ 4 & 6 & 1 & 2 \\ 6 & 4 & 2 & 1 \end{bmatrix} \times \begin{bmatrix} S0(X_{i,0} \oplus RK_{i,0}) \\ S1(X_{i,1} \oplus RK_{i,1}) \\ S0(X_{i,2} \oplus RK_{i,2}) \\ S1(X_{i,3} \oplus RK_{i,3}) \end{bmatrix}$$

where $S0$ and $S1$ are CLEFIA's 8×8 s-boxes. $X_{i,k}$ and $Z_{i,k}$ represent a byte of the input and output word respectively ($0 \leq k \leq 3$), while $RK_{i,k}$ is a byte of the round key. The matrix multiplication is in a finite field.

Consider four look-up tables as follows which map an input byte (x) to a 32-bit word.

$$\begin{aligned} T_0[x] &\leftarrow (6 \cdot S0(x)|4 \cdot S0(x)|2 \cdot S0(x)|1 \cdot S0(x)) \\ T_1[x] &\leftarrow (4 \cdot S1(x)|6 \cdot S1(x)|1 \cdot S1(x)|2 \cdot S1(x)) \\ T_2[x] &\leftarrow (2 \cdot S0(x)|1 \cdot S0(x)|6 \cdot S0(x)|4 \cdot S0(x)) \\ T_3[x] &\leftarrow (1 \cdot S1(x)|2 \cdot S1(x)|4 \cdot S1(x)|6 \cdot S1(x)). \end{aligned}$$

Then,

$$\begin{aligned} (Z_{i,3}|Z_{i,2}|Z_{i,1}|Z_{i,0}) = &T_0[X_{i,0} \oplus RK_{i,0}] \oplus T_1[X_{i,1} \oplus RK_{i,1}] \\ &\oplus T_2[X_{i,2} \oplus RK_{i,2}] \oplus T_3[X_{i,3} \oplus RK_{i,3}] \end{aligned}$$

The function $F1$ is defined as follows:

$$F1: \begin{bmatrix} Z_{i+1,0} \\ Z_{i+1,1} \\ Z_{i+1,2} \\ Z_{i+1,3} \end{bmatrix} = \begin{bmatrix} 1 & 8 & 2 & A \\ 8 & 1 & A & 2 \\ 2 & A & 1 & 8 \\ A & 2 & 8 & 1 \end{bmatrix} \times \begin{bmatrix} S1(X_{i+1,0} \oplus RK_{i+1,0}) \\ S0(X_{i+1,1} \oplus RK_{i+1,1}) \\ S1(X_{i+1,2} \oplus RK_{i+1,2}) \\ S0(X_{i+1,3} \oplus RK_{i+1,3}) \end{bmatrix}$$

The function F1 can be implemented using four tables as shown below. Each table maps the 8-bit input x to a 32-bit output.

$$\begin{aligned} T_4[x] &\leftarrow (A \cdot S1(x)|2 \cdot S1(x)|8 \cdot S1(x)|1 \cdot S1(x)) \\ T_5[x] &\leftarrow (2 \cdot S0(x)|A \cdot S0(x)|1 \cdot S0(x)|8 \cdot S0(x)) \\ T_6[x] &\leftarrow (8 \cdot S1(x)|1 \cdot S1(x)|A \cdot S1(x)|2 \cdot S1(x)) \\ T_7[x] &\leftarrow (1 \cdot S0(x)|8 \cdot S0(x)|2 \cdot S0(x)|A \cdot S0(x)) \end{aligned}$$

Then,

$$\begin{aligned}(Z_{i+1,3}|Z_{i+1,2}|Z_{i+1,1}|Z_{i+1,0}) = & T_4[X_{i+1,0} \oplus RK_{i+1,0}] \\ & \oplus T_5[X_{i+1,1} \oplus RK_{i+1,1}] \\ & \oplus T_6[X_{i+1,2} \oplus RK_{i+1,2}] \\ & \oplus T_7[X_{i+1,3} \oplus RK_{i+1,3}].\end{aligned}$$

The output of the round is

$$\begin{aligned}(Y_{i,0} \cdots Y_{i,3}) &\leftarrow (Z_{i,0} \cdots Z_{i,3}) \oplus (X'_{i,0} \cdots X'_{i,3}) \\ (Y'_{i,0} \cdots Y'_{i,3}) &\leftarrow (X_{i+1,0} \cdots X_{i+1,3}) \\ (Y_{i+1,0} \cdots Y_{i+1,3}) &\leftarrow (Z_{i+1,0} \cdots Z_{i+1,3}) \oplus (X'_{i+1,0} \cdots X'_{i+1,3}) \\ (Y'_{i+1,0} \cdots Y'_{i+1,3}) &\leftarrow (X_{i,0} \cdots X_{i,3}).\end{aligned}$$

In a similar manner, all rounds of CLEFIA can be implemented. Thus reducing the implementation to a series of memory accesses intertwined with ex-ors.

2.2.3 CAMELLIA

CAMELLIA is the 128-bit block cipher that was jointly developed by Mitsubishi and NTT in 2000 [8]. Since the cipher has been made available under a royalty-free license, it has been certified for use by the European Union and Japan. It has also become part of the OpenSSL project, and incorporated in Mozilla's Network Security Services (NSS modules). Support for CAMELLIA has been added to several security libraries as well as Mozilla's popular web browser, Firefox.

The 128-bit block cipher CAMELLIA has a Feistel structure as shown in Fig. 2.6. The 16 bytes plaintext input is grouped in two words of 8 bytes each: $\mathbf{x} = (x_1 \| x_2 \| \cdots \| x_8)$ and $\mathbf{y} = (y_1 \| y_2 \| \cdots \| y_8)$. There are 18 rounds in all, broken up into groups of 6 each. After the 6th and the 12th rounds, there are two FL/FL^{-1} function layers. In each round, there is an F function, which is a combination of key addition, substitution (S), and permutation (P). The substitution is done by using four s-boxes, whereas, the P function is implemented by using a diffusion matrix. Figure 2.7 shows the permutation operation and the diffusion matrix. The diffusion matrix also has an inverse depicted in the figure.

Each round has an addition of a round key. The ith round uses the round key $\mathbf{k}^{(i)}$. Each of these round keys are of 64 bits. Additionally, whitening keys $kw1$ and $kw2$ are applied at the start of encryption, while $kw3$ and $kw4$ are applied at the end of encryption.

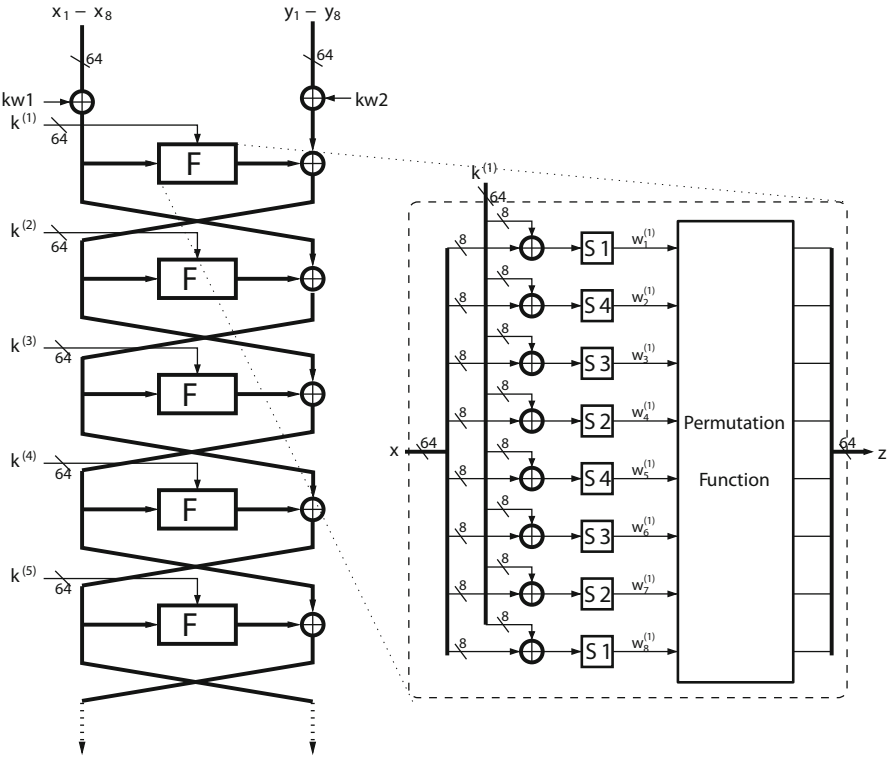


Fig. 2.6 Structure of CAMELLIA

Fig. 2.7 CAMELLIA's diffusion layer and its inverse

$$\begin{pmatrix} z_8 \\ z_7 \\ \cdot \\ \cdot \\ z_1 \end{pmatrix} \leftarrow P \bullet \begin{pmatrix} w_8 \\ w_7 \\ \cdot \\ \cdot \\ w_1 \end{pmatrix}$$

(a) Permutation Function

$$\begin{pmatrix} 0&1&1&1&1&0&0&1 \\ 1&0&1&1&1&1&0&0 \\ 1&1&0&1&0&1&1&0 \\ 1&1&1&0&0&0&1&1 \\ 0&1&1&1&1&1&1&0 \\ 1&0&1&1&0&1&1&1 \\ 1&1&0&1&1&0&1&1 \\ 1&1&1&0&1&1&0&1 \end{pmatrix}$$

(b) Diffusion Matrix P

$$\begin{pmatrix} 1&1&1&0&1&0&0&1 \\ 0&1&1&1&1&1&0&0 \\ 1&0&1&1&0&1&1&0 \\ 1&1&0&1&0&0&1&1 \\ 0&1&1&1&0&1&1&1 \\ 1&0&1&1&1&0&1&1 \\ 1&1&0&1&1&1&0&1 \\ 1&1&1&0&1&1&1&0 \end{pmatrix}$$

(c) Inverse Diffusion Matrix P^{-1}

2.3 Classical Cryptanalysis

A cryptanalytic attack is a procedure through which an attacker gains information about the secret decryption key. Attacks are classified according to the level of a priori knowledge available to the attacker.

A *ciphertext-only attack* is an attack where the cryptanalyst has access to ciphertexts generated using a given key but has no access to the corresponding plaintexts or the key. A *known-plaintext attack* is an attack where the cryptanalyst has access to both ciphertexts and the corresponding plaintexts but not the key.

A **chosen-plaintext attack** (CPA) is an attack where the cryptanalyst can choose plaintexts to be encrypted and has access to the resulting ciphertexts, again their purpose being to determine the key.

A **chosen-ciphertext attack** (CCA) is an attack in which the cryptanalyst can choose ciphertexts, apart from the challenge ciphertext and can obtain the corresponding plaintext. The attacker has access to the decryption device.

In case of CPA and CCA, adversaries can make a bounded number of queries to its encryption or decryption device. The encryption device is often called an oracle; meaning it is like a black-box without details like in an algorithm of how an input is transformed or used to obtain the output. Although this may seem a bit hypothetical, but there are enough real life instances where such encryption and decryption oracles can be obtained.

In the next section, we discuss one form of classical cryptanalysis of block ciphers, namely differential attacks, which is used in developing some of the cache attacks described in the book.

2.3.1 Classical Cryptanalysis of Block Ciphers

Block ciphers have been subjected to several forms of classical cryptanalysis; namely linear cryptanalysis, differential cryptanalysis, impossible differential attacks, related key attacks, boomerang attacks, square attacks are some popular cryptanalysis techniques. In this book, we study how microarchitectural features like cache memories and the accompanying hardware increases leakage when an encryption program runs on this platform. These attacks, often called as side-channel attacks (SCA) exploit additional leakage through timing information, and combine with classical methods, like differential attacks to provide efficient attack techniques. In this section, we provide a quick overview on the ideas of differential cryptanalysis, which will be useful for other attacks. Interested readers may refer [9] for ideas on other forms of cryptanalysis, which may (or may not) lead to more efficient cache attacks.

2.3.2 The Idea of Differential in Block Ciphers

Differential cryptanalysis is a chosen plain text attack and the attack is based on the information of the ex-or of two inputs, and the ex-or of corresponding outputs. Consider, a cipher expressed as $c = m \oplus k$, where m, k, and c are the plaintext, key, and the ciphertext blocks respectively, each of size b-bits. We know this is a secured cipher if the key is randomly and uniquely chosen for every encryption. The attacker has no information about the plaintext from the ciphertext. However, if the key is used twice for two encryptions there is a leakage of information. It is trivial to note, $c_0 \oplus c_1 = m_0 \oplus m_1$, thus showing that the *differential* (or, the difference which is computed useing ex-ors between two blocks) does not depend on the key, and thus the key (although unknown) does not hide the plaintext from the attacker! The differential is often denoted by $\Delta(c) = c_0 \oplus c_1$. Thus, in brevity we can state

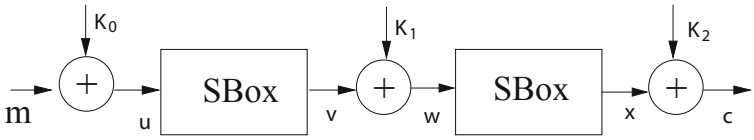


Fig. 2.8 Illustration of a differential attack on a toy cipher

Table 2.1 The S-Box description

x	0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F
$s[x]$	C	5	6	B	9	0	A	D	3	E	F	8	4	7	1	2

$\Delta(c) = \Delta(m)$ for the above equality. It may be noted that if the key mixing was done by some other reversible process (like modular integer addition), the differential would be accordingly modified (by using modular integer subtraction). Let us see how we can use this fact to cryptanalyze block ciphers with S-Boxes.

In the Fig. 2.8, a two round encryption is depicted where the plaintext m is transformed by mixing (ex-oring) with the keys, k_0, k_1, and k_2 along with substitution which occurs due to the nonlinear S-Boxes. Let us denote the S-Box mapping by S, and thus an input x is transformed to an output which is $S[x]$. The mapping we choose for illustration is that of the block cipher PRESENT[10]. For convenience we present the mapping in Table 2.1.

From the notion of differentials we know that ex-oring with the key has no effect on the differential. However, the S-Box being a nonlinear layer prevents passing of the ex-or differential. Thus, the attacker can guess k_2 and obtain the value of x, for two different encryptions. We denote this by saying that c_0 gives x_0, while c_1 gives x_1. One can perform the inverse S-Boxes (which have to be invertible!), and obtain w_0 and w_1. However, the uncertainty of k_1 prevents us from computing the values of v_0 and v_1. However, we know $\Delta(w) = w_0 \oplus w_1 = v_0 \oplus v_1 = \Delta(v)$. Likewise, if the attacker chooses $\Delta(m) = m_0 \oplus m_1$, she can compute $\Delta(u)$, however the S-Box prevents from determining $\Delta(v)$.

So, differential cryptanalysis performs a differential analysis of the S-Box. Consider, the Table 2.2, where the input differential $i \oplus j$, for two inputs i and j, is *chosen* to be F. The output differential, $S[i] \oplus S[j]$ is computed, and its frequency is observed. It may be seen that some values do not occur in the output differential. Good design practice in the PRESENT S-Box ensures that the probability distribution is uniform, meaning the differentials, namely E, 4, 1, and F which occurs in the output occurs with the same probability, i.e., $\frac{1}{4}$. The attacker can thus choose the values of m_0 and m_1, such that $\Delta(m) = F$, and thus $\Delta(u) = F$. Due to the differential property of the S-Box we know that in 4 out of 16 cases, the output differentials can be E, 4, 1, and F. Thus, $\Delta(w)$ is also any one of the above choices. The attacker also guesses k_2, and obtains the value of $\Delta(w)$ by inverting the S-Box on the values x_0 and x_1, obtained by guessing the key k_3. It is clear that those guesses which provide $\Delta(w)$ any other value, but E, 4, 1, F, can be eliminated as wrong keys. However, the differential property of the S-Box cannot distinguish the keys which lead to $\Delta(w)$

Table 2.2 Differential analysis of the S-Box

i	j	$S[i]$	$S[j]$	$S[i] \oplus S[j]$
0	F	C	2	E
1	E	5	1	4
2	D	6	7	1
3	C	B	4	F
4	B	9	8	1
5	A	0	F	F
6	9	A	E	4
7	8	D	3	E
8	7	3	D	E
9	6	E	A	4
A	5	F	0	F
B	4	8	9	1
C	3	4	B	F
D	2	7	6	1
E	1	1	5	4
F	0	2	C	E

equal to the above four values. This approximately eliminates 3/4 of the keys! This simple example shows the power of differential cryptanalysis, and can be used to distinguish the wrong keys from the correct ones.

In real life the block ciphers are cascades of substitution and permutations, and thus often referred to as SPN (Substitution Permutation Network) ciphers. However, the above discussion of differential cryptanalysis can be conceptually applied even for such a construction. In such a case, for a four-round cipher as shown in Fig. 2.9, an input–output differential is found for three rounds of the cipher which occurs with a very high probability. It is often defined as a differential trail, $\Delta_i \rightarrow \Delta_o$, where Δ_i and Δ_o are the input differential and output differential after the third round. The output differential also should ensure it is of low weight, meaning that it disturbs or affects less number of S-Boxes of the final round. The attacker can then guess a part of the last round key, corresponding to the disturbed S-Boxes, and then decrypt a portion of the ciphertext to check the differential at the output of the third round. It is expected that for the correct key the differential should be equal to Δ_o, with a very high probability. It may be noted that the efficiency of such an attack compared to a brute force attack is because it is a divide-and-conquer strategy, and thus can be successful by guessing portions of the key.

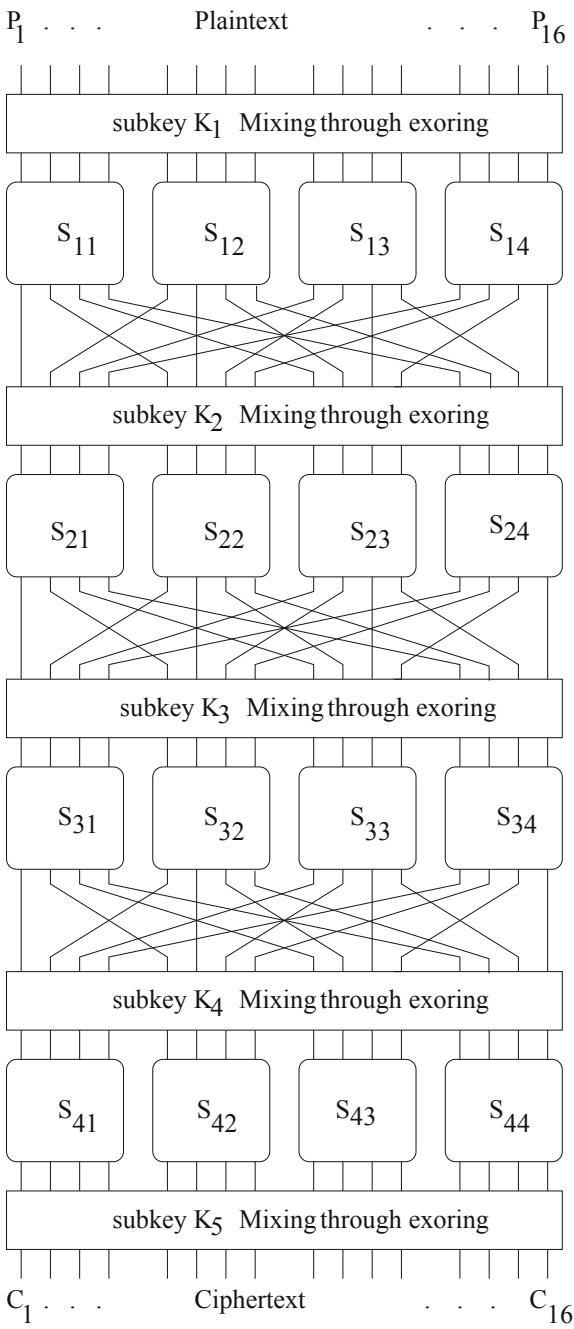


Fig. 2.9 The SPN block cipher

2.4 Asymmetric-Key Ciphers

The book also deals with attacks on asymmetric-key ciphers also known as public-key ciphers. These algorithms are often computation intensive and operate on large finite fields. For efficient design several field operation algorithms have been developed for supporting multiplication, inverse, exponentiation etc., and have been implemented in software libraries. The next section provides a quick overview on some of the commonly known techniques which are employed to realize the ciphers, and are hence targeted for demonstrating timing attacks. We focus on RSA, as it is still one of the most popular and utilized public-key algorithms. Further more, many of the attack techniques that we discuss in the book can be extended to other well known public-key algorithms, like elliptic curve cryptosystems.

2.5 RSA: An Asymmetric-Key Algorithm

RSA works by considering two keys: a *public key* is known to every one whereas a *private key* is *secret*. Encryption of a message is performed using the public key, but decryption requires the knowledge of the private key. All the operations are done mod n, where n is the product of two large distinct prime numbers p and q. The values of p and q are, however, private and hence not disclosed to all. The encryption key, which is public is a value b, where $1 \leq b \leq \phi(n)$ and $\phi(n)$ is the Euler's totient function. Very simply put, Euler's totient function or phi function, $\phi(n)$, is an arithmetic function that counts the positive integers less than or equal to n that are relatively prime to n. The decryption key is a private value a, which is selected such that $ab \equiv 1 \, mod \, \phi(n)$. The owner of the private key (p, q, a) publishes the value (b, n), which is the public key.

The encryptor chooses a message x, where $x \in Z_n$. It may be mentioned that $Z_n = \{0, 1, \ldots, n-1\}$. The encryption process is computing the cipher as $y \equiv x^b \, mod \, n$ using the public key b. Since the decryptor knows the value of a, which is the private key, he computes the value of x from y by computing $y^a \equiv (x^b)^a \, mod \, n \equiv x \, mod \, n$.

Algorithm 2.3: Square and Multiply algorithm for Exponentiation

Input: Ciphertext: C, Modulus: N and Private key: $k = (k_{m-1}, k_{m-2} \cdots k_0)$, where $k_{m-1} = 1$
Output: Plaintext: $M \equiv C^k \bmod N$

```
1 R = C
2 for i = m − 2 to 0 do
3 |   R = R^2 mod N
4 |   if k_i = 1 then
5 |   |   R = R × C mod N
6 |   end
7 end
8 return R
```

The security of *RSA* is based on the assumption that decryption can be performed only by the knowledge of the private key b. However, to obtain the private information from the public value a requires one to compute the modular inverse of a modulo $\phi(n)$. It is believed that to obtain $\phi(n)$ from n requires the knowledge of the prime factors of n, namely p and q. The security of RSA is thus based on the hardness assumption of factorization of large n. Thus, the underlying operation to perform RSA encryption and decryption is modular exponentiation, i.e., $y \equiv x^b \, mod \, n$, where b is typically 1024 bits or 2048 bits large. This is achieved by the popular square and multiply algorithm.

2.5.1 Square and Multiply Algorithm to Perform Exponentiation

In this section, we present the square and multiply algorithm to perform modular exponentiation. We present the decryption algorithm, which uses the decryption key, denoted as $k = a$. We focus on the decryption algorithm as that is a natural target for an attacker, as the secret key is involved in the computation. The key is an m-bit key, denoted as $k = (k_{m-1}, k_{m-2} \cdots k_0)$, where $k_{m-1} = 1$.

This algorithm, as we later elaborate in the book, is naturally vulnerable against SCA. The reason being that the operations, namely squaring and multiplication, have different fingerprints on the side-channel leakage. For example, with respect to a timing attack, both requires different number of clock cycles to execute. We can observe that if a key bit is one, then a multiplication is performed, else not. The attacker hence tries to exploit this conditional property on the key bits to devise an attack. In the literature, there are several ways of performing a square and multiply algorithm to achieve exponentiation. One of the most popular techniques is called Montgomery Ladder, which is explained in the Algorithm 2.4.

It may be observed that in the Montgomery's Ladder, irrespective of the key bit, a multiplication and squaring is always performed. Thus, the design is naturally resistant against some SCA, which are possible over the naïve square and multiply algorithm.

Algorithm 2.4: Montgomery's Ladder for Exponentiation

Input: Ciphertext: C, Modulus: N and Private key: $k = (k_{m-1}, k_{m-2} \cdots k_0)$, where $k_{m-1} = 1$
Output: Plaintext: $M \equiv C^k \bmod N$

```
P1 = C, P2 = C^2 mod N
for i = m − 2 to 0 do
    if k_i = 1 then
        P1 = P1 × P2 mod N, P2 = P2^2 mod N
    end
    else
        P2 = P1 × P2 mod N, P1 = P1^2 mod N
    end
end
return P1
```

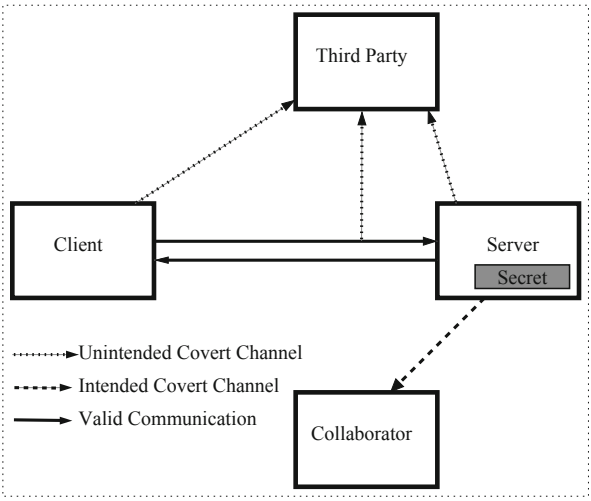


Fig. 2.10 Lampson's confinement problem

2.6 Confinement Problem and Covert Channels

Now that we have reviewed the essential components in cryptography, we move to side channels and how they covertly leak information about an executing application.

Consider a client using the services of a server. The client provides an input which is operated on by the server using some stored secret. In an untrusted environment, the client should ensure that the server does not communicate its input to a collaborator (the attacker), while the secret stored in the server should not be revealed to the client. Additionally no third-party should gain any information about the transactions between the client and server. This is the *confinement problem* as defined by Lampson in 1973 [10] and pictorially represented in Fig. 2.10.

Generally, the system can ensure that the server does not communicate with the collaborator by disabling writing to files, shared memories, and other inter-process communication (IPC) protocols. The server's secret can be protected by implementing memory protection using schemes such as paging and access control. In spite of these protection schemes, there still exists indirect paths by which data can be communicated. These indirect paths are not meant for communication, hence known as *covert channels*. They are of interest in the domain of computer security because they allow programs to bypass security policies of the system. They are all the more relevant in the cloud computing environments where such untrusting parties sharing resources are common. Covert channels can either be *intended* (where a nexus exists between the server and the collaborator) or *unintended* (where a third party obtains information about the computations). Covert channels are generally noisy, but information theory can be used to devise an encoding, which will allow data to get through reliably no matter how small the signal is, provided it is not zero [10].

An example of a covert channel was illustrated by Schaefer et al. in [11]. Consider the server and collaborator sharing a CPU. The client can signal information to the

collaborator by the amount of time it holds the CPU. For instance, holding the CPU for 10, 20, or 30 μs can be used to represent a 0, 1, or 2 respectively [11]. This forms an *intentional* covert channel between the two cooperating processes (the client and the collaborator). Over the years several covert channels have been discovered such as the rate at which a program performs paging [10, 12], CPU scheduling [13], disk scheduling [13, 14], and cache memory usage [15, 16].

Covert channels do not always have to be intentional. There can also be unintentional covert channels, which results in transfer of information about a program execution to a third party. The program is unaware of these transfers. Such unintentional covert channels occur due to physical attributes of a device such as the power consumption and electromagnetic radiation, as well as execution time. These unintentional covert channels are commonly known as *side channels*. In 1996, Kocher showed that a timing side channel can be used to reveal the secret key of a cipher to an adversary [17]. Later, power consumption of the device was used for the same purpose [18]. These works that came to be known as SCA (side-channel attacks), kindled interest in the academic community and led to a new domain of cryptographic research; targeting implementations of ciphers rather than just the algorithm. The next section surveys SCA against ciphers that use unintentional covert timing channels. This class of attacks is called *timing attacks*.

2.7 Formal Analysis of Side-Channel Attacks

Since Shannon's benchmark paper in [19], there has been significant progress in the mathematical treatment given to cryptography. Ciphers, such as the three discussed in the previous section, underwent rigorous analysis with strong attack models such as adversaries who know the cipher algorithm, the input, as well as the output. The only secret being the cipher's key. All ciphers standardized and in use today have bounded security against these attack models.

The advent of SCA in [17] introduced a stronger (yet practical) attack model. Here, the adversary has access to side-channel information leakage of the encrypting device in addition to the input, output, and cipher algorithm. The mathematical tools developed so far failed to model these attack classes. It was not until 2004 that new models were developed by Micali and Reyzin to analyze cryptography in the presence of side-channel leakage [20]. This they called *physical observable cryptography*. The new theory was enhanced by the works of Standaert [21–24] and by Backes and Kopf [25, 26, 27] and used to provide a fair evaluation and comparison of ciphers in the presence of side-channel leakage. This has led to the development of a new class of cryptographic algorithms and countermeasures with provable security in the presence of leakages (for example [28, 29, 30]). The algorithms currently available are inefficient to implement. The hope is that the tools and models will lead to practically realizable ciphers with provable resistance against SCA. In this section we present briefly the formal notions in side-channel analysis.

Let $E_{\mathbf{k}}$ be a cipher having a secret key $\mathbf{k}$ chosen uniformly from the set $\mathscr{K}^m$ for some positive integer m. Let $\widetilde{E}_{\mathbf{k}}$ be an implementation of $E_{\mathbf{k}}$ on a device. A *q-limited side-channel key recovery adversary* is a statistical program, which can make at-most q queries to $\widetilde{E}_{\mathbf{k}}$ and monitor the leakage through channels such as power-consumption, timing, or electromagnetic radiation of the device. To quantify leakage, a *leakage function* $L(\cdot)$ is defined that mathematically abstracts the characteristics of the side channel and the measurement setup. For example, it is well-known that power consumption can be modeled in terms of Hamming weight or Hamming distances for CMOS devices.

The q-limited side-channel key recovery adversary follows a divide-and-conquer strategy by splitting $\mathbf{k}$ into smaller parts for example $\mathbf{k} = (k_1 \| k_2 \| k_3 \| \cdots \| k_m)$, where $k_1, k_2, \cdots, k_m \in \mathscr{K}$, and each key part is targeted independently. Further, the leakage partitions the key space $\mathscr{K}$ into equivalence classes such that the SCA cannot distinguish between two keys in the same class. The goal of the adversary is to guess a key class with nonnegligible probability. To do so, typically the attack has two phases. An online phase in which the q side-channel leakages are collected and an offline phase in which the leakages are analyzed using statistical distinguishers in order to obtain a ranking of the keys in $\mathscr{K}$ in an order of their likelihood.

There are two metrics by which the success of the attack can be measured. The first defines the oth order *success rate* as the probability that the correct key is ranked among the top o keys. For example, if $\mathscr{G} = (g_1, g_2, \cdots g_o, g_{o+1}, \cdots, g_{|\mathscr{K}|})$ is the ordered sequence of keys ranked from most likely to least likely, then the probability that the correct value of the key is in $g_1, g_2 \cdots, g_o$ is the oth order success rate of the attack. Alternatively, *guessing entropy* [31] can be used as a metric to evaluate the success of an attack. This measures the average number of guesses that are required before obtaining the correct key. For example, if the correct key is present in the jth location in the ranking $\mathscr{G}$ then the guessing entropy is j. The second metric uses information-theoretic metrics to determine for example $\mathbb{H}[K|L]$, i.e., the entropy of the key given the leakage. This should be much less than $\mathbb{H}[K]$ for a strong attack.

This book shows how timing attacks can be modeled by analyzing the cipher's memory access patterns. The mathematical framework thus developed is used to evaluate ciphers for their resistance against timing attacks and choose implementation strategies for the ciphers.

2.8 Conclusion

This chapter provided an overview of various symmetric and asymmetric encryption schemes. Classical cryptanalytic techniques were dwelt upon, and side channels and their formal analysis were introduced. To a side-channel attacker, it is not just the enciphering algorithm and their implementations that are important, but the system and CPU architecture is also crucial. The next chapter provides an overview of modern CPUs and the few components in them that have been used to develop SCA.

References

1. Gueron S (2010) Intel Advanced Encryption Standard (AES) instructions set (Rev : 3.0)
2. Zheng Y, Matsumoto T, Imai H (1989) On the construction of block ciphers provably secure and not relying on any unproved hypotheses. In: Brassard G (ed) CRYPTO. Lecture notes in computer science, vol 435. Springer, Berlin, pp 461–480
3. Federal Information Processing Standards Publication 197 (2001) Announcing the Advanced Encryption Standard (AES)
4. Stinson D (2002) Cryptography: theory and practice, 2nd edn. Chapman and Hall, London, pp 117–154
5. Daemen J, Rijmen V (2002) The design of Rijndael: AES—the Advanced Encryption Standard. Springer, Berlin
6. Shirai T, Shibutani K, Akishita T, Moriai S, Iwata T (2007) The 128-bit blockcipher CLEFIA (extended abstract). In: Biryukov A (ed) FSE Lecture notes in computer science, vol 4593. Springer, Berlin, pp 181–195
7. Sony Corporation (2007) The 128-bit blockcipher CLEFIA : algorithm specification
8. Aoki K, Ichikawa T, Kanda M, Matsui M, Moriai S, Nakajima J, Tokita T (2000) Camellia: A 128-bit block cipher suitable for multiple platforms—design and analysis. In: Stinson DR, Tavares SE (eds) Selected areas in cryptography. Lecture notes in computer science, vol 2012. Springer,Berlin, pp 39–56
9. Knudsen LR, Robshaw MJB (2011) The block cipher companion. Springer, Berlin
10. Lampson BW (1973) A note on the confinement problem. Commun ACM 16(10):613–615. http://doi.acm.org/10.1145/362375.362389
11. Schaefer M, Gold B, Linde R, Scheid J(1977) Program confinement in KVM/370. In: Proceedings of the 1977 annual conference, ser. ACM '77. ACM: New York, pp. 404–410. http://doi.acm.org/10.1145/800179.1124633. Accessed Dec 2013
12. Van Vleck T (1990) Timing channels. Multics, technical report, 1990
13. Kemmerer RA (1983) Shared resource matrix methodology: an approach to identifying storage and timing channels. ACM Trans Comput Syst 1(3):256–277. http://doi.acm.org/10.1145/357369.357374
14. Karger PA, Wray JC (1991) Storage channels in disk arm optimization. IEEE symposium on security and privacy. IEEE, Oakland, pp 52–63
15. Wray JC (1991) An analysis of covert timing channels. In: Research in security and privacy, 1991. Proceedings, 1991 IEEE computer society symposium. May 1991, pp 2–7
16. Hu W-M (1992) Lattice scheduling and covert channels. In: Proceedings of the 1992 IEEE symposium on security and privacy. SP '92. IEEE Computer Society, Washington, DC, pp 52–61
17. Kocher PC (1996) Timing attacks on implementations of Diffie-Hellman, RSA, DSS, and other systems. In: Koblitz N (ed) CRYPTO '96: proceedings of the 16th annual international cryptology conference on advances in cryptology. Lecture notes in computer science, vol 1109. Springer-Verlag, London, pp 104–113
18. Kocher PC, Jaffe J, Jun B (1999) Differential power analysis. In: Wiener MJ (ed) CRYPTO. Lecture notes in computer science, vol 1666. Springer, Berlin, pp 388–397
19. Shannon CE (1949) Communication theory of secrecy systems. Bell Syst Tech J 28:656–715
20. Micali S, Reyzin L (2004) Physically observable cryptography (extended abstract). In: Naor M (ed) TCC. Lecture notes in computer science, vol 2951. Springer, Berlin, pp 278–296
21. Standaert F-X, Peeters E, Archambeau C, Quisquater J-J (2006) Towards security limits in side-channel attacks. In: Goubin L, Matsui M (eds) CHES. Lecture notes in computer science, vol 4249. Springer, Berlin, pp 30–45
22. Standaert F-X, Pereira O, Yu Y, Quisquater J-J, Yung M, Oswald E (2009c) Leakage resilient cryptography in practice. Cryptology ePrint archive, Report 2009/341. http://eprint.iacr.org/. Accessed June 2010

23. Standaert F-X, Malkin T, Yung M (2009b) A unified framework for the analysis of side-channel key recovery attacks. In: Joux A (ed) EUROCRYPT. Lecture notes in computer science, vol 5479. Springer, Berlin, pp 443–461
24. Standaert F-X, Koeune F, Schindler W (2009a) How to compare profiled side-channel attacks? In: Abdalla M, Pointcheval D, Fouque P-A, Vergnaud D (eds) ACNS. Lecture notes in computer science, vol 5536, pp 485–498
25. Köpf B, Basin DA (2007) An information-theoretic model for adaptive side-channel attacks. In: Ning P, di Vimercati SDC, Syverson PF (eds) ACM conference on computer and communications security. ACM, Alexandria, pp 286–296
26. Backes M, Köpf B (2008) Formally bounding the side-channel leakage in unknown-message attacks. In: Jajodia S, ópez JL (eds) ESORICS. Lecture notes in computer science, vol 5283. Springer, Berlin, pp 517–532
27. Backes M, Köpf B, Rybalchenko A (2009) Automatic discovery and quantification of information leaks. In: IEEE symposium on security and privacy. IEEE Computer Society, 2009, pp 141–153
28. Dziembowski S, Pietrzak K (2008) Leakage-resilient cryptography. In: FOCS. IEEE Computer Society, 2008, pp 293–302
29. Naor M, Segev G (2009) Public-key cryptosystems resilient to key leakage. In: Halevi S (ed) CRYPTO. Lecture notes in computer science, vol 5677. Springer, Berlin, pp 18–35
30. Pietrzak K (2009) A leakage-resilient mode of operation. In: Joux A (ed) EUROCRYPT. Lecture notes in computer science, vol 5479. Springer, Berlin, pp 462–482
31. Massey J (1994) Guessing and entropy. In: Information theory, 1994. Proceedings, 1994 IEEE international symposium, 1994, p 204

Chapter 3
Superscalar Processors, Cache Memories, and Branch Predictors

Memory accesses and branches are arguably the biggest performance bottlenecks in a program. To mitigate their effect on the performance, modern superscalar microprocessors incorporate cache memories and branch predictors in their architecture. While cache memories bridge the performance gap between the processor and the main memory, branch predictors make predictions about branch destinations to reduce the overhead of branches in the program. The pitfall of these components is that they result in information leakage through side channels, which have been used to break crypto-systems. This book provides an analysis of the attacks that occur due to these components. This chapter provides a brief overview of superscalar processors, its memory organization, cache memories, and branch predictors [1–3].

3.1 Superscalar Processors

In this section, superscalar processors and features such as speculation and out-of-order execution are discussed. Figure 3.1 shows a block diagram of a superscalar processor. A superscalar processor can be thought of as a pipeline with several stages; each stage doing a specific job. In every clock cycle, multiple instructions are fetched from the instruction cache into an instruction/decode buffer. The dispatch unit then scrutinizes the instructions in the buffer and attempts to issue them to an appropriate functional unit (such as integer arithmetic logic units (ALU), floating point units (FPU), branch unit, etc.). The issue is done provided the functional unit is available and the operands used by the instruction are up-to-date. It may be noted that the instructions can be issued to the functional units out-of-order (i.e., in an order that does not strictly follow the program flow). Consequently, the result from the functional unit is obtained out-of-order. This is stored in a reorder buffer before it is retired (i.e., the processor registers updated). An instruction is retired if and only if all instructions preceding it have retired. Instructions can even also be executed speculatively, that is, before a branch can be taken.

Figure 3.2 shows an example of how instructions generally flow in a processor having a five stage pipeline. In first stage (IF), instructions are fetched from the

C. Rebeiro et al., *Timing Channels in Cryptography,*
DOI 10.1007/978-3-319-12370-7_3

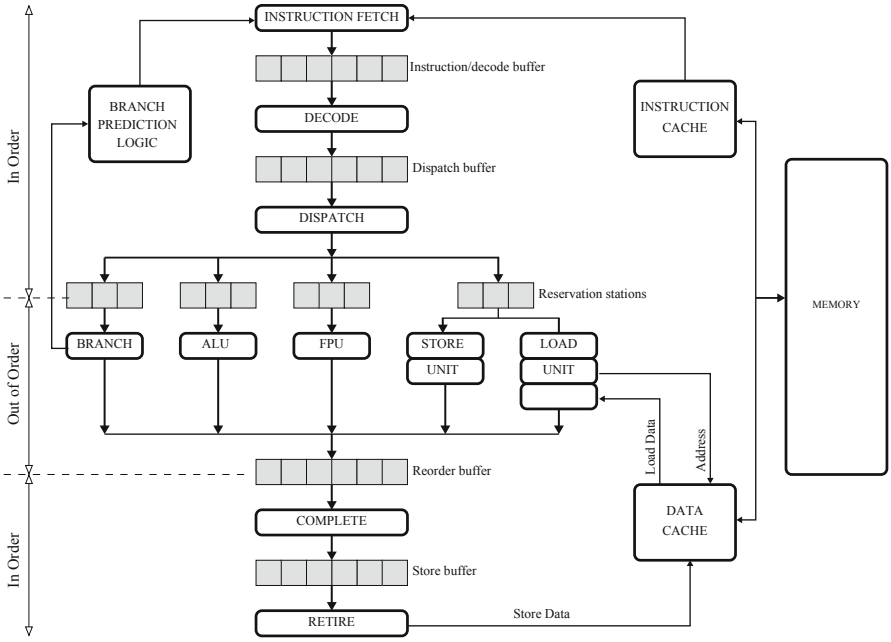


Fig. 3.1 Superscalar processor architecture

instruction cache. The second stage (ID) decodes instructions. The third stage (OF), fetches operands required, while the fourth stage (IE) executes the instructions. The result of the instructions are written back in final stage (WB). Several instructions are simultaneously operated upon in the processor. While one is in the IE stage, others are in the IF, ID, OF, and WB stages. Each stage is small and completes its task fast. This allows the processor to operate at high clock frequencies.

Modern processors can have over 30 pipeline stages. This deeply pipelined execution is possible because instructions are generally executed in a predictable sequence. The processor can, therefore, fetch and decode instructions much ahead of time. This nice flow in the instruction stream is broken by branches in the program. Branch instructions break the sequential flow requiring instructions to be executed from some arbitrary branch destination. A branch instruction requires the entire pipeline to be flushed. All instructions in various stages of execution would need to be aborted and new instructions from the branch destination fetched. This has significant overheads because as many as 100 instructions could concurrently be present in different execution stages of the pipeline. Branch predictors are used to reduce this overhead by predicting branch target destinations and avoid flushing the pipeline. Section 3.3 has more details.

A functional unit of interest in timing attacks is the load-store unit (especially the load part), which handles all memory accesses made by the program. Each load or store has three operations. The first is an address generation (which generates the

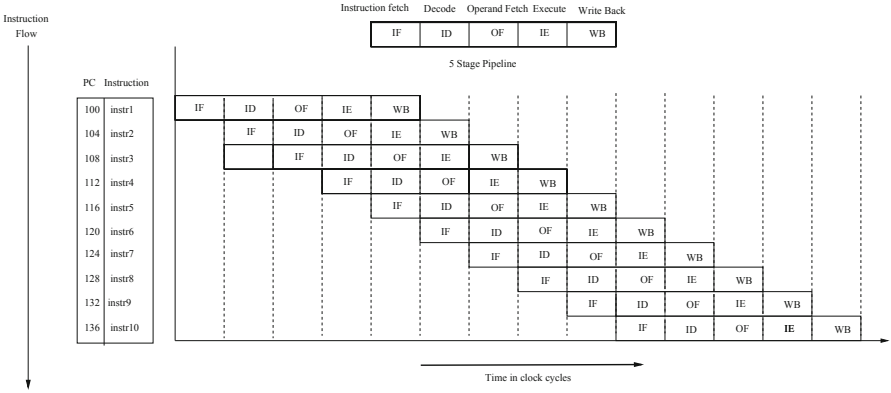


Fig. 3.2 Instruction flow in a five-stage pipeline. The program counter (*PC*) points the instruction being executed

effective address from where data are to be loaded or stored). Next, the address is translated from a virtual address to a physical one using a translation look aside-buffer (TLB). The final operation either loads or stores data to or from the memory. For a load instruction, the loaded data are updated into a register only after the ordering has completed. Load instructions can be pipelined, executed speculatively, and out-of-order. Additionally, multiple load instructions can be issued in parallel if the processor supports multiple load-store units.

In order to extract the best performance out of a superscalar processor, the memory subsystem must match the performance of the processor. The next section briefly describes memory subsystems and branch prediction logic in superscalar processors.

3.2 Memory Hierarchy and Cache Memory

Modern superscalar CPU architectures are plagued with the *von Neumann bottleneck* due to the high speed with which CPUs process instructions and the comparative low speed of memory access. The possibility of using high-speed memory technologies that match CPU processing speeds is restricted by cost factors. To mitigate this speed difference, superscalar CPU architectures are built with memory as a hierarchy of levels, with small and fast memories close to the processor and large, slower, less expensive memories below it. A memory stores a subset of the data that is present in the memory below it, while the lowest memory level (generally main memory that uses low-cost capacitive DRAMs) contains all the data. The aim is to provide the user with large cheap memories, while providing access at processor speeds.

Memories close to the processor are called *cache memories* and are built with SRAM technology, which offers high access speeds, but are large and expensive. Depending on the data it contains, cache memories are categorized as either *data*, *instruction*, or *unified* (that store both data and instructions). Cache memories are

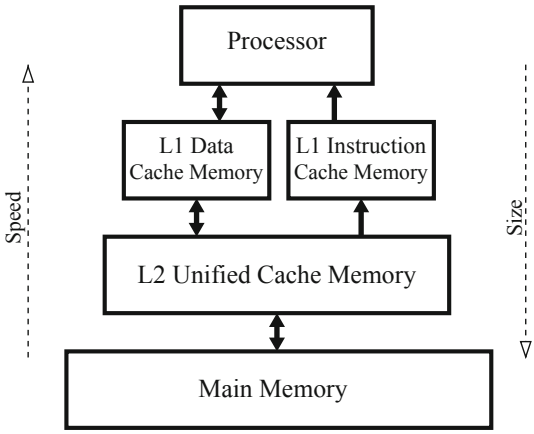


Fig. 3.3 Memory hierarchy structure

also categorized based on their closeness to the processor as L1, L2, and L3, with L1 being the closest to the processor and L3 the farthest. Most modern desktop and server processors have dedicated L1 data and L1 instruction caches, and a larger unified L2 cache memory. Some servers also contain an L3 cache before the main-memory (Fig. 3.3). The L1 caches data present in L2, while L2 caches data present in L3 or the main memory.

Cache memories work by exploiting the *principle of locality*, which states that programs access a small portion of their address space in any given time instant. The entire memory space of the processor is divided into *blocks*, typically of 64 or 128 contiguous bytes. A memory access results in a *cache hit* if the data are available in the cache memory. For example, an *L1-data cache hit* is obtained if the data accessed are available in the L1-data cache memory. In this case, the data are available to the processor quickly and there are no delays. A *cache miss* occurs if the data accessed are not available in the cache memory. In this case the block containing the data is loaded into the cache from a memory lower in the hierarchy. This is a slow process during which the processor may stall. There are three types of cache misses: compulsory misses, capacity misses, and conflict misses. *Compulsory misses* are cache misses caused by the first access to a block that has never been used in the cache. *Capacity misses* occur when blocks are evicted and then later reloaded into the cache. This occurs when the cache cannot contain all the blocks needed during execution of a program. *Conflict misses* occur when one block replaces another in the cache.

3.2.1 Organization of Cache Memory

The cache memory is organized into lines with each line capable of holding a block of memory. Suppose each block has a capacity to store 2^{δ} words, and there are 2^{b} lines in the cache, then at most $2^{b+\delta}$ words can be stored in the cache. An address

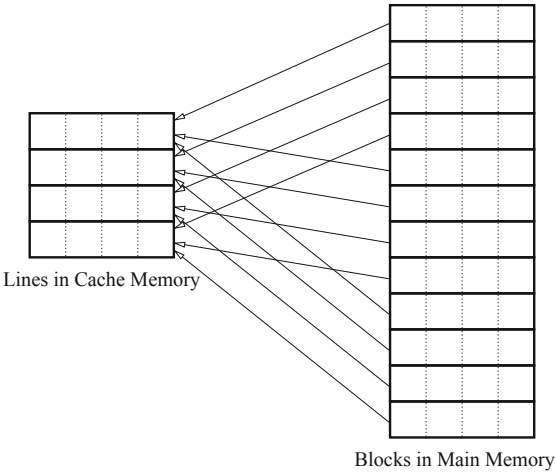


Fig. 3.4 Direct mapped cache memory with $2^b = 4$

translation mechanism is required to determine which cache line a block should get stored into. In the most simple approach, every 2^b-th memory block gets mapped into the same cache line. This mapping is called *direct-mapping* and is depicted in Fig. 3.4.

The address translation mechanism computes two components: the word address A_{word} and the line address A_{line}. If A is the address of the data that is to be accessed then

$$A_{word} = A \bmod 2^{\delta}$$
$$A_{line} = \lfloor A/2^{\delta} \rfloor \bmod 2^{b} .$$

Note that the size of the address A_{word} is δ bits while the size of the line address is b bits. Problems arise due to the many-to-one mapping from blocks to lines (as seen in Fig. 3.4). The cache controller needs to know which block is present in a cache line before it can decide if a hit or a miss has occurred. This is done by using an identifier called the *tag*. The identifier denoted A_{tag} for the address A is given by

$$A_{tag} = \lfloor \lfloor A/2^{\delta} \rfloor / 2^{b} \rfloor .$$

Thus, every line in the cache has an associated tag as shown in Fig. 3.5. For every memory access, the tag for the address (A_{tag}) is compared with the tag stored in the cache. A match results in a cache hit, otherwise a cache miss occurs.

Time-driven cache attacks on block ciphers monitor these hits and misses that occur due to look-up tables used in the implementation. A look-up table is defined as a global array in a program. For example, the C construct for a look-up table is as follows.

```
const unsigned char T0[256]={0x63, 0x7C, 0x77, ... };
```

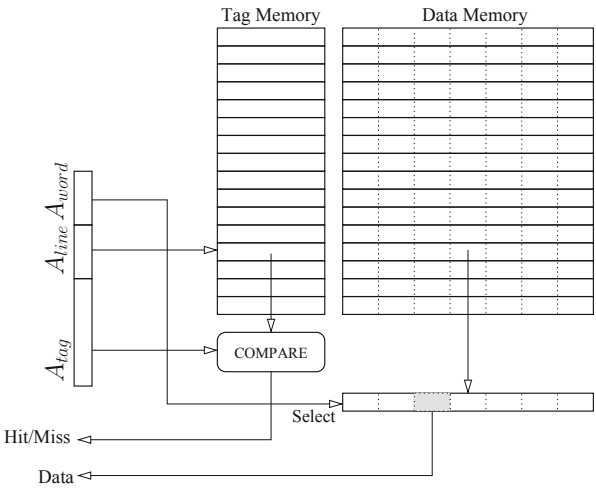


Fig. 3.5 Organization of direct mapped cache memory

Assuming that the base address for `T0` is `0x804af40`, Table 3.1 shows how the table gets mapped for a $4KB$ direct-mapped cache with a cache line size of 64 bytes. Every block in the table maps to a distinct cache line provided that the table has lesser blocks than the lines in the cache.

Table 3.1 Mapping of Table `T0` to a direct-mapped cache of size $4KB$ ($2^{\delta} = 64$ and $2^{b} = 64$)

Elements	Address	line	Tag
`T0[0]` to `T0[63]`	`0x804af40` to `0x804af7f`	`61`	`0x804a`
`T0[64]` to `T0[127]`	`0x804af80` to `0x804afbf`	`62`	`0x804a`
`T0[128]` to `T0[191]`	`0x804afc0` to `0x804b0ff`	`63`	`0x804a`
`T0[192]` to `T0[255]`	`0x804b000` to `0x804b03f`	`0`	`0x804b`

The drawback of the direct mapped scheme is poor cache utilization and *cache thrashing*. Consider a program which continuously reads data from addresses A and then A'. The addresses are such that they map into the same line in the cache (i.e., $A_{line} = A'_{line}$). Thus every memory access would result in a cache miss, considerably slowing down the program even though the other lines in the cache are unused. This is called cache thrashing.

An improved address translation scheme divides the cache into 2^s sets. Each set groups $w = 2^b/2^s$ cache lines. A block now maps to a set instead of a line and can be present in any of the w cache lines in the set. A cache that uses such an address translation scheme is called a *w−way set associative cache*. The address translation for such a scheme is defined as follows,

$$A_{word} = A \bmod 2^{\delta}$$
$$A_{set} = \lfloor A/2^{\delta} \rfloor \bmod 2^{s}$$
$$A_{tag} = \lfloor \lfloor A/2^{\delta} \rfloor / 2^{s} \rfloor$$

If $w = 2$, the thrashing in the previous example is eliminated as the data corresponding to the addresses A and A' can simultaneously reside in the cache. However, the problem arises again if a program is executed that makes continuous accesses to three or more data from different addresses that share the same cache set.

3.2.2 Improving Cache Performance for Superscalar Processors

As seen in the previous section, a poorly designed cache can slow down a program instead of accelerating it. Therefore, it is important to understand the effect of the cache memory in a program's execution. The average memory access time is often used as a metric for the purpose. For a system with a single level of cache memory, the metric is defined as follows,

$$\text{Average memory access time} = \text{Hit time} + \text{Miss Rate} \times \text{Miss Penalty},$$

where, Hit time is the time to hit in the cache, while Miss Penalty is the time required to replace the block from memory (i.e., the miss time). Miss Rate is the fraction of the memory accesses that miss the cache.

Improving cache performance can be done by reducing hit time, reducing or hiding miss penalty, and/or reducing miss rate. To certain extent basic parameters in the cache design can be tuned to improve cache performance. The cache size ($2^{b+\delta}$), for example, should be large enough to provide temporal locality for the program, yet be small enough to prevent extensive degradation of hit time. Similarly, other cache parameters such as the associativity (w), block size (2^b), and number of levels can be tuned.

Besides tuning the basic parameters in the cache, other micro-architectural techniques are available to improve performance of cache memories. These are generally incorporated in modern superscalar microprocessors. The remaining of the section summarizes some of the important techniques.

- *Wide cache interfaces.* The data bus interface in these caches is capable of transferring multiple contiguous words at a time. For example, two words to an entire line can be transferred simultaneously. This helps in reducing multicycle accesses, thus reducing on average the hit time. Wide cache interfaces are especially suited for instruction caches and L2 and L3 level cache memories, where this feature allows the entire cache line to be transferred efficiently.
- *Multiaccess cache.* Single access cache memories can handle one read or one write per clock cycle. This becomes a bottleneck in superscalar processors, which are capable of multiple loads or stores in a clock cycle. In such cases multiaccess cache

memories are an advantage as they allow multiple accesses in a clock cycle, thus preventing stalls in the processor. There are various ways in which multi-access cache memories can be realized. We enumerate some of the methods.

- *True Multiporting.* Uses dual-ported memory to store the tags and data. However the size of the cache increases significantly and the hit time is also affected.
- *Multiple Cache Copies.* Mirrors cache memories so that each mirror holds an identical copy of the data and can handle a single access. Thus n mirrors can support n accesses simultaneously. In addition to the large size, a significant drawback is that stores require all mirrors to be updated in order to maintain coherency.
- *Overclocking.* The cache is clocked x times faster than the processor. Therefore, a single cycle of the processor can service x accesses to the cache.
- *Multibanked caches.* The address space is partitioned into multiple banks. Bank 1 caches addresses from partition 1, Bank 2 caches addresses from partition 2, and so on. Each bank can service one memory access per clock cycle. Thus, if there are n partitions, n memory accesses can be serviced together provided they go to different banks. Two simultaneous accesses to the same bank causes a *conflict* and results in a stall. Additionally a crossbar switch is required to channel requests to the appropriate bank and another crossbar switch is required to channel the result to the appropriate port.

- *Sub-blocks.* The cache line is broken into several sub-blocks. Each sub-block has a separate valid bit. On a cache miss, only the required sub-block needs to be fetched. Similarly, for stores, only the valid sub-blocks need to be stored. The others can be ignored thereby reducing the miss penalty.
- *Critical word first.* During a cache miss, the required word in the block is fetched first, thus allowing the processor to restart early. This reduces the miss penalty.
- *Write Buffers.* A write buffer is used to hold data before it is written back into the cache. This reduces the latency for the write and also allows the cache to service other read requests while the store is taking place.
- *Non-blocking or lockup free caches.* In conventional caches while a miss is being serviced, a second access to the cache is stalled. One remedy is to buffer future misses, while allowing the cache to continue servicing hits. A better alternative is to allow multiple misses to be overlapped. This is a nonblocking cache that helps increasing the cache bandwidth.
- *Pipelined caches.* This allows a new request to be serviced before the previous is completed. Thus increasing the cache bandwidth.
- *Victim Caches.* These are small fully associative caches that store blocks recently evicted from the L1 cache. They are accessed on a miss in parallel with the lower level cache. This avoids lower level cache access and reduces the number of cache misses as it is fully associative.
- *Prefetching Cache.* If programs follow strictly the principle of locality, then cache misses can be considerably reduced. However, most programs occasionally access data that are not spatially local. This will create additional cache misses. One

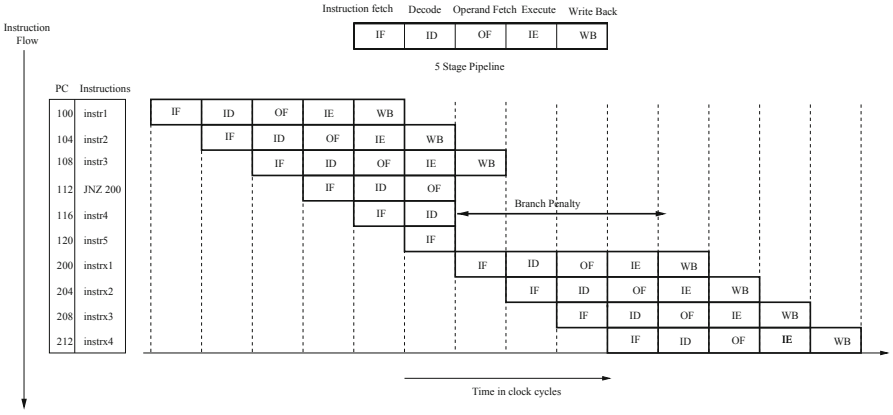
remedy is to load data into the cache before it is actually required, thereby reducing the miss rate. This is called prefetching and can be done either in software or automatically by the hardware. *Software prefetching* requires that prefetch instructions be inserted into the program that would initiate a load of a block into the cache. However, unlike normal loads, the instruction does not stall.
Hardware prefetching uses a prefetcher that monitors the memory access patterns of the executing program and predicts the future data the program will access. This datum is loaded into the cache before the program accesses the data, thus reducing the miss latency. Hardware prefetchers are categorized as *sequential prefetchers* and *stride prefetchers* depending on the sequence of memory accesses. A sequential prefetcher tracks consecutive blocks accessed by the program, while a stride prefetcher looks for accesses with regular strides that are not necessarily consecutive.
Implementing a hardware prefetcher requires a memory reference prediction table, comprising of three fields. The first is the instruction address of the load, which is used as a tag field for selecting an entry of the table. The second field contains the previous instruction address of the load, while the third has the stride value. In order to prefetch, the previous address is added to the stride value in order to obtain a predicted address. If the data corresponding to the predicted address is not in the cache, it is loaded from the memory lower in the hierarchy.

3.3 Branch Prediction Unit

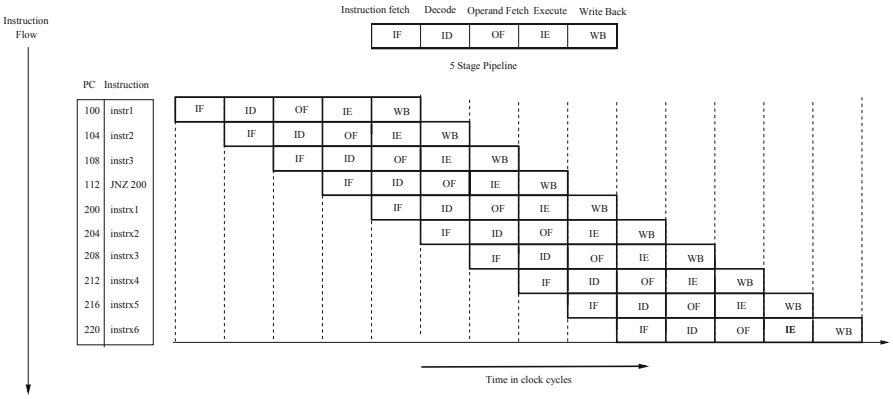
A branch in a program causes its control flow to deviate from a sequential execution. Branch instructions can either be unconditional or conditional. In an unconditional branch instruction, the control flow of the program changes from its sequential flow unconditionally. The next instruction executed is not the next in the instruction sequence. In the case of a conditional branch instruction, the control flow changes only if the condition specified in the instruction is satisfied. The conditions are evaluated only when the branch instruction reaches the execute stage of the processor. By then, several more instructions have entered the processor pipeline and are in various stages of execution. If the branch is not taken, execution flows normally and no overheads are incurred. However, if the branch is taken, all instructions in the pipeline would need to be flushed and new instructions from the branch destination be fetched. This leads to significant overheads.

The Branch Prediction Unit (BPU) plays an important role in reducing this overhead. Much before the branch condition is evaluated, the BPU guesses the probable execution path of the program and causes the processor to fetch instructions from the branch target. This avoids the pipeline flush, saving critical clock cycles. The prediction is made based on historic information about the branches in the program. Figure 3.6 shows the effect of the branch predictor in saving clock cycles.

Like all predictors, the BPU's prediction is not always correct. It could result in a misprediction, where the target address is not the same as the predicted target.



a Without a Branch Predictor



b With a Branch Predictor

Fig. 3.6 Branch predictors can reduce clock cycles by predicting the branch target. The figures show an example of the working of a branch predictor in a processor with a five-stage pipeline. The first column in the figures (100,104,...) are the memory addresses, while the second are the instructions (instr1, instr2, ... instrx1, instrx2,....). Without the predictor, the *JNZ* (jump if not zero) instruction causes the pipeline to be flushed and four clock cycles wasted. On the other hand, a processor with a branch predictor that predicts the branch target (PC = 200) correctly would save these clock cycles

This would cause overheads as the pipeline needs to be flushed. An efficient branch prediction algorithm is one where such mispredictions are minimized.

There has been considerable research in developing the "ideal" branch prediction algorithm; one which has no mispredictions. The algorithms developed are based on either static or dynamic schemes. The next section discusses these schemes.

3.3.1 Static Branch Prediction

Some of the static branch prediction schemes are listed below:

- Whenever a branch instruction is present, predict that the branch is always taken. Alternatively, the prediction could always be not taken.
- Predict depending on the opcode. For instance, opcode X would always result in a prediction that the branch is taken, while opcode Y would always result in a prediction that the branch is not taken.
- Branches that jump forward in the program are always predicted not taken, while branches that jump back are always taken.

The drawback of these approaches is that they do not perform well when the executing program has hierarchical loop statements or when there is an irregularity in the branch behavior. In such situations, the dynamic branch prediction strategies perform better.

3.3.2 Dynamic Branch Prediction Schemes

Dynamic prediction schemes store significant amount of information on the run-time instruction execution and use this information to predict the outcome of branch instructions. This section describes some of the most common dynamic predictors.

3.3.2.1 1-bit Branch predictor

This predictor is also known as *last-time* predictor and is arguably the simplest form of dynamic branch prediction. Every time a branch instruction is executed, the predictor saves the result—either taken or not taken. This stored result is used to make a prediction the next time the branch instruction is fetched.

The predictor hardware requires a table containing the address of the branch instruction and a bit, which signifies branch taken or not taken. This bit is updated every time the branch instruction is executed. The automaton for the one bit predictor is shown in Fig. 3.7. The automaton has two states—one predicts branches to be taken and the other predicts branches to be not taken. The transitions are marked by the output of the last branch statement.

3.3.2.2 Bimodal Predictor

In the 1-bit branch predictor discussed above, every misprediction would change the result of the prediction. In a bimodal predictor on the other hand, two consecutive mispredictions are required to change the result of the prediction. Figure 3.8 shows

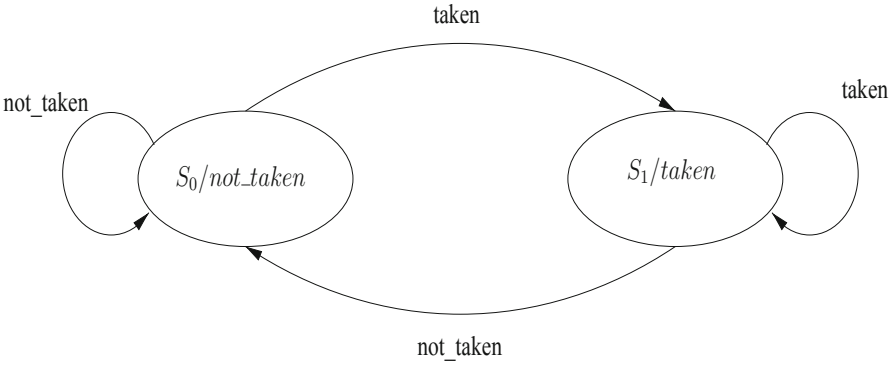


Fig. 3.7 Dynamic 1-bit predictor state machine

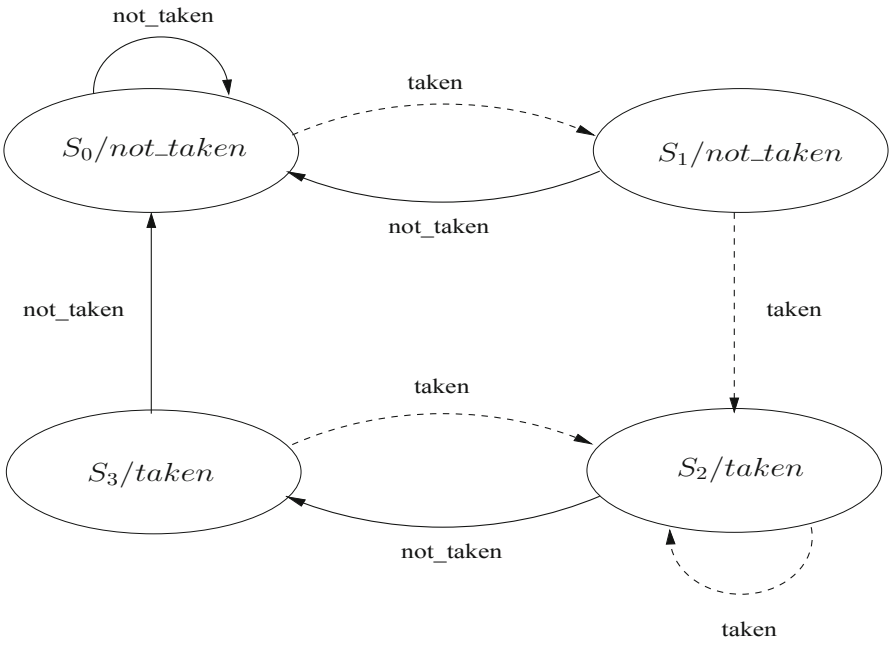


Fig. 3.8 Dynamic 2-bit predictor state machine

the state machine for a bimodal predictor (also called a 2-bit predictor), which predicts branches as either taken or not taken.

The state machine has four states S_0, S_1, S_2, and S_3, each having a predicted output. The arrows represent whether a branch is taken or not, while the label inside the state denotes the prediction (which is the output of the predictor). The original Intel Pentium used this form of prediction.

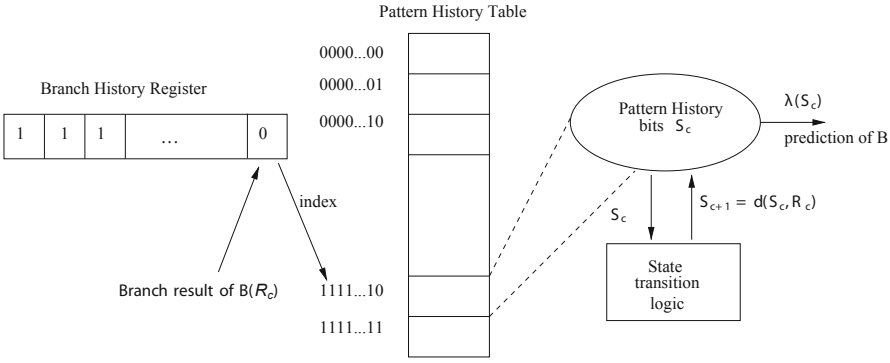


Fig. 3.9 Two level adaptive branch prediction

3.3.2.3 Two-Level Adaptive Predictor

Conditional branches that are taken in a regular recurring pattern are not predicted well by the bimodal predictor. In such cases a two-level adaptive predictor works better. The predictor remembers the last k occurrences of a branch instruction and uses an s-bit prediction function (such as an s-bit predictor) for each of the 2^k history of patterns. The first level in the two level adaptive predictor uses a *branch history register*. This is a shift register of size k, which stores the history of the last k branches—either taken or not taken. The branch history register indexes into a second level called *pattern history table*, which can hold 2^k entries, each s bits wide. An s-bit prediction function is used to make a prediction based on the last s branches. Figure 3.9 illustrates the two level predictor.

When a conditional branch say $\mathscr{B}$ is getting predicted,

- content of the k bit history register is used as address to the pattern history table. Let S_c be the contents of the pattern history table.
- S_c is fed to the s-bit prediction decision function (such as the s-bit predictor), which outputs the predicted value $\lambda(S_c)$.

To take an example, consider the case when k = 2, which means that the last two occurrences of the branch are stored in the branch history register. The register therefore takes 4 values (00, 01, 10, and 11), which indexes into the 0th, 1st, 2nd, and 3rd entry in the pattern history table. The contents of the pattern history table is fed to a bimodal predictor.

Now consider a branch sequence is 0011001100. Further, assume that s is 2 and uses a 2 bit predictor function. The 00 entry would predict 1, the 01 entry would predict 1, the 10 entry would predict 0, and the 11 entry would predict 0.

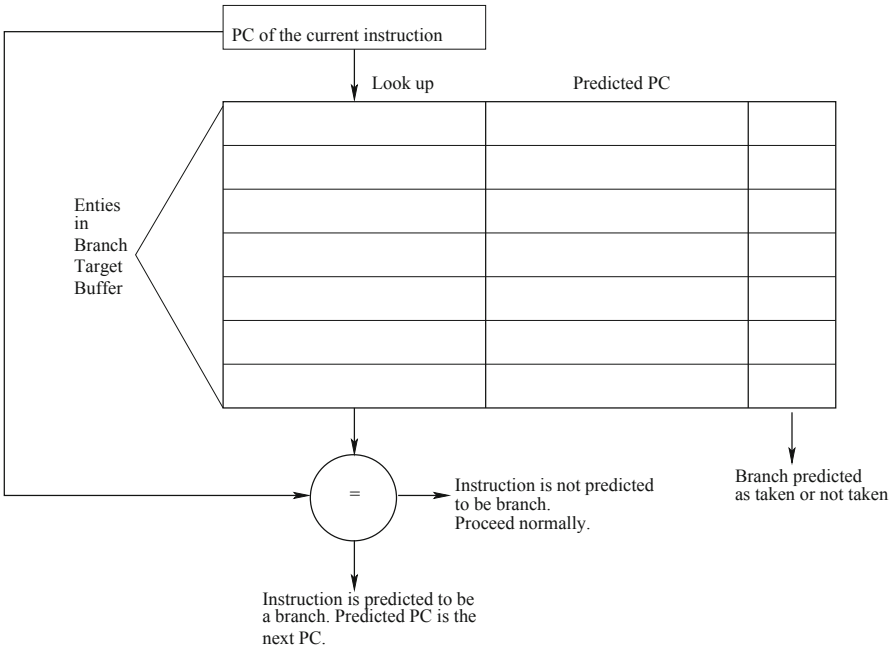


Fig. 3.10 The branch target buffer in a processor

3.3.3 Branch Target Buffers

Branch target buffers (BTB) is a cache in the processor that stores branch targets. Each BTB entry is primarily composed of tag bits, the target address of the branch, status bits indicating if the branch is taken or not taken. Whenever a branch instruction is encountered, a part of the instruction address (PC—program counter) is used as an index to the BTB location.

The BTB has limited size and thus can hold a limited number of branch targets. If a new branch instruction is encountered that is not in the BTB, then an entry in the BTB would be evicted. Figure 3.10 shows the structure of the BTB.

3.4 Conclusion

This chapter laid the foundation for superscalar processors, cache memories, and branch prediction units. Timing attacks discussed from here on would rely significantly on how these components operate when a cipher executes. In the next chapter, we would demonstrate this by showing how the cache memories affect the cipher execution and how they can be exploited to reveal information about the cipher's secret key.

Reference

1. Hennessy JL, Patterson DA (2006) Computer architecture: a quantitative approach, 4th ed. Morgan Kaufmann
2. Shen JP, Lipasti MH (2005) Modern processor design: Fundamentals of superscalar processors. McGraw-Hill
3. Juan T, Navarro JJ, Temam O (1997) Data Caches for Superscalar Processors, in International Conference on Supercomputing, pp 60–67

Chapter 4
Time-Driven Cache Attacks

When a load instruction incurs a cache-miss, a block of memory from the lower level of the memory subsystem is loaded into a cache line. Consequently, the memory access would require considerably more time and power, and has a characteristically different electromagnetic radiation compared to when a cache-hit occurs. Figure 4.1, for instance, shows the power consumption trace for eight memory load instructions on a PowerPC processor. The loads that result in cache misses are easily distinguishable from the cache hits. These indirect manifestations of a memory access can be used by an attacker to gain considerable insight about the application currently being executed. In this chapter, we show how information about the secret key of a cipher can be gleaned from the execution time of a block cipher. We start the chapter with a simple illustration showing how information can be obtained from memory access patterns before discussing attacks on ciphers.

4.1 A Simple Illustration

Consider a program having a look-up table T and a function called `AccessT`, which takes as input two parameters x_1 and x_2. The function accesses the table T twice—at locations $k_1 \oplus x_1$ and $k_2 \oplus x_2$ as shown in Listing 4.1, where k_1 and k_2 are a pair of application specific secret data.

Suppose a malicious user in the system wants to obtain information about the secrets k_1 and k_2. However, she can only invoke `AccessT` and does not have sufficient privileges to observe the internals of `AccessT`. For instance, `AccessT` can be a system call executing in the operating system, while the malicious user has only user level permissions. Since no direct channels to determine k_1 and k_2 are present, the malicious user could resort to using indirect channels. For instance, the user could invoke `AccessT` and observe the power consumption while the function executes. These side-channels would indirectly provide information about the accesses made to T.

Assuming that the cache memory does not contain any part of the look-up table before the first invocation of `AccessT`, the first access at an offset $(x_1 \oplus k_1)$ in the look-up table would result in a cache miss. Consequently, a memory block containing

C. Rebeiro et al., *Timing Channels in Cryptography,*
DOI 10.1007/978-3-319-12370-7_4

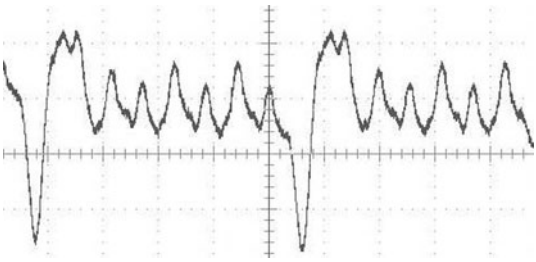


Fig. 4.1 The power consumption trace of eight load instructions in a PowerPC processor. The first and the fifth load resulted in a cache miss, while the others were cache hits

the data $T[x_1 \oplus k_1]$ would get loaded into a cache line. For instance, if the size of the cache line is 32 bytes and each element in the table requires 1 byte then at most $32 = (32/1)$ elements of the look-up table are loaded into the cache.

Listing 4.1 function AccessT reads two locations from a look-up table

```
unsigned char T[256] = {0×33, 0×65, ···};
unsigned char y;

unsigned char AccessT(unsigned char x1, unsigned char x2){
  unsigned char k1 = load_secret1();
  unsigned char k2 = load_secret2();

  y = T[k1 ⊕ x1] ⊕ T[k2 ⊕ x2];

  return y;
}
```

During the second access to table T at an offset $(x_2 \oplus k_2)$, a cache hit will arise if (1) data corresponding to the first access still resides in the cache and (2) the access at offsets $(x_1 \oplus k_1)$ and $(x_2 \oplus k_2)$ in the look-up table fall in the same memory block. We call this a *collision*. This cache hit can be inferred by monitoring the power consumption of the device. The cache hit implies that $(x_1 \oplus k_1)$ and $(x_2 \oplus k_2)$ differ by at most 32 (for a cache line of 32 bytes). This indicates that $(x_1 \oplus k_1)$ and $(x_2 \oplus k_2)$ differ by $\log_2 32 = 5$ least significant bits. We represent this relationship as $\langle x_1 \oplus k_1 \rangle = \langle x_2 \oplus k_2 \rangle$, where $\langle z \rangle$ is $z \gg 5$. Since the malicious user chooses x_1 and x_2, a relationship between k_1 and k_2 can be inferred as follows,

$$\langle k_1 \oplus k_2 \rangle = \langle x_1 \oplus x_2 \rangle \; . \tag{4.1}$$

For instance, if x_1 and x_2 are chosen as $0{\times}45$ and $0{\times}c4$, respectively and the memory block size is 32 bytes, then $\langle k_1 \oplus k_2 \rangle$ is 4. This is obtained as follows,

$$\begin{aligned} \langle k_1 \oplus k_2 \rangle &= (x_1 \oplus x_1) \gg \log(32/1) \\ &= (0{\times}45 \oplus 0{\times}c4) \gg 5 \\ &= 4 \end{aligned} \tag{4.2}$$

This means that the value of the three most significant bits of $k_1 \oplus k_2$ is 4. The set of candidates are therefore $\{0\times80, 0\times81, 0\times82, \cdots 0\times9f\}$. Only one value in the candidate set is correct, all others are wrong.

In terms of the information gained, since k_1 and k_2 are defined as unsigned char in Listing 4.1, each can take one of 2^8 different values. Together, the uncertainty of k_1 and k_2 is 2^{16} (or 16 bits). After observing the cache hit, the uncertainty of k_1 and k_2 reduces. In the example taken above, with the 32 byte memory block size, the uncertainty of k_1 and k_2 reduces to 2^{13} (or 13 bits).

The amount by which the uncertainty reduces depends on the size of the look-up table, the memory block size which is equivalent to the cache line size, and the alignment of the table in memory. This reduction in uncertainty is discussed in Sects. 4.1.1 and 4.1.2.

A cache miss could also be a source providing information about k_1 and k_2. A cache miss in the second table access would indicate that the first and second memory access to table T are to different memory blocks. Thus , $\langle x_1 \oplus k_1 \rangle \neq \langle x_2 \oplus k_2 \rangle$ and

$$\langle k_1 \oplus k_2 \rangle \neq \langle x_1 \oplus x_2 \rangle \ . \tag{4.3}$$

In Listing 4.1, a cache miss in the second access would eliminate 32 values for $k_1 \oplus k_2$.

A cache miss, however, can also occur if the memory block corresponding to $T[x_1 \oplus k_1]$ gets evicted from the cache before $T[x_2 \oplus k_2]$ is accessed. Thus unlike a cache hit, which guarantees a collision, a cache miss does not always imply that a collision has not occurred. Said differently, Eq. 4.3 may not always hold even if a cache miss has occurred during the second access to table T.

4.1.1 Relation Between Size and Bits Revealed

The size of the memory block, look-up table, and the size of each element in the table dictate the number of bits that get revealed. If N is the number of elements in the look-up table, L the size of each memory block in terms of bytes and B the number of bytes required to store each element in the look-up table, then the number of elements in a memory block is L/B and the number of memory blocks occupied by the table is $(N \times B)/L$.

Power consumption traces of a cache hit determine collisions in a memory block. Thus number of bits revealed is $\log_2(\frac{N\times B}{L})$. Since multiple elements of the table share the same block, there is an ambiguity about which element was actually accessed within the block. This results in multiple candidates of size $\frac{L}{B}$. In Listing 4.1, N is 256, B is 1 (since sizeof(unsigned char) = 1), and assume L to be 32. Thus, the number of bits of $k_1 \oplus k_2$ revealed is 3 and the number of candidates for $k_1 \oplus k_2$ is 32. The number of bits revealed reduces as the size of the look-up table reduces. The ideal case of zero bits revealed occurs when the look-up table fits within a single memory block.

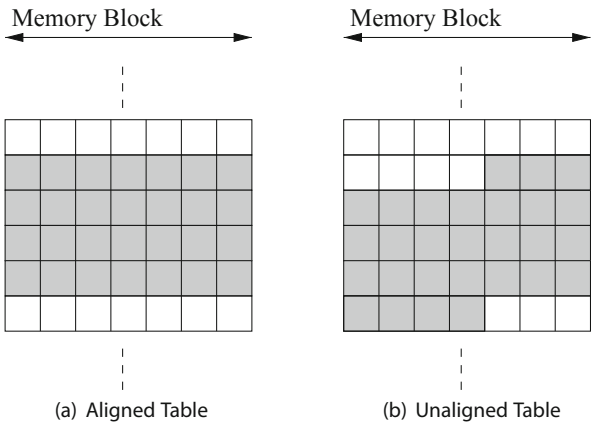


Fig. 4.2 The alignment of a table in memory could affect the bits revealed. Each row in the figure is a memory block, while the *shaded regions* are the look-up table of size l. In the aligned table every block has δ elements, while in an unaligned table the first and last block have fewer elements

4.1.2 Relation Between Alignment of Tables and Bits Revealed

A look-up table is said to be *aligned* if its base address is a multiple of the memory block size. Figure 4.2a shows how an aligned table is placed in memory, while Fig. 4.2b shows an *unaligned* look-up table. Each row in the figures indicates a memory block. The alignment of the table depends considerably on the compiler used to build the program's executable. Some compilers align tables in order to achieve better performance, while others do not align tables thus utilizing memory better. If the tables are dynamically allocated, then the operating system's memory management scheme dictates the table's alignment.

In an aligned table, all memory blocks loaded into the cache contain the same number of elements of the look-up table. Thus the size of the candidate key set is always L/B. This size is independent of which memory block is loaded into the cache. In an unaligned table, the first and the last memory block of the table contain lesser number of elements compared to the other blocks. Consequently, a cache hit in the second access occurring due to a collision in either the first or the last memory block would result in a candidate key set which is smaller than L/B.

For instance, let A be a memory address which is a multiple of the memory block size. Assume that the table T in Listing 4.1 has the base address $A + 31$. Since the table is of size 256 bytes, it occupies 8 memory blocks assuming each block is of 32 bytes. The first memory block contains 1 element of the table, the last block contains 15 elements, while all other memory blocks holding the table contain 16 elements. Consequently, the size of the candidate key set would be 1, 16, or 15 depending on which memory block had the collision.

4.1.3 Initial State of Cache Memory

In the discussion above, we assumed that no part of table T was present in the cache before function `AccessT` was first invoked. In order to ensure this, the attacker could

clear the cache of the contents of T before invoking `AccessT`. Some processors have instructions that would allow parts of the cache to be flushed. Intel and AMD x86 processors, for instance, have an instruction called clflush for the purpose. The instruction takes a linear address and invalidates lines in all caches that contain the address. Alternatively the attacker could fill the cache with its own data, thus evicting all other data. This is done by allocating an array as large as the cache itself and then accessing it. The access would load the array elements into the cache, thereby evicting all other data.

4.2 Collisions from Execution Time

The previous section showed how power consumption of a processor can be used to detect cache hits and misses. Power consumption side-channels are generally suited for small embedded processors such as the PowerPC, where tapping into the power lines is easy. In more complex processors, the noise in the power consumption would significantly hinder the determination of a collision. Time-based side-channels are generally preferred in such environments as they can be easily measured, do not require physical colocation with the device, and do not require any special equipment.

In a timing side-channel microarchitectural events such as a cache hit and a miss are distinguished using time measurements. Making these measurements requires a highly precise clock source. Many microprocessors have such precise clocks in the form of a time stamp counter. This is a highly precise low-overhead way to make time measurements. In processors that do not have such counters, virtual time-stamp counters (VTSCs) have been built to measure microarchitectural events. VTSCs have a small loop that increments a counter continuously. Since the loop is small, the counter is incremented at every clock cycle of the processor. The value of the counter, therefore, gives a notion of time. Such ad hoc clocks are feasible because time is only used to distinguish between microarchitectural events, the exact duration of the events are not required. For instance, time side-channels just use the fact that a cache miss takes more time than a cache hit. The exact duration of a cache miss and cache hit is not important. In this section we discuss these two clock sources.

4.2.1 Clocks Using Hardware Time Stamp Counters

Intel machines since the Pentium have a 64-bit time-stamp counter register. The register is made 0 when the processor resets and is monotonically incremented in every clock cycle. Thus, on a 1 GHz machine, the counters could measure time periods as low as a nanosecond. An instruction called rdtsc is used to read the time-stamp counter. When rdtsc is invoked, the high-order 32 bits of the counter is loaded into the edx register, while the low-order 32 bits is loaded into the eax register. The following pseudoassembly code shows how the rdtsc instruction can be used to time a load instruction.

Listing 4.2 *timestamp* (using rdtsc to measure the time required for a load instruction)

```
1 rdtsc                ; read time stamp
2 mov    time, eax     ; move counter into variable
3 load   ebx, (ebp)    ; a load from memory
4 rdtsc                ; read time stamp again
5 sub    eax, time     ; find the difference
```

The load instruction in line 3 loads the ebx register with contents pointed to by the register ebp. The load instruction is sandwiched between two rdtsc instructions. The first rdtsc records the counter value in the variable time before the load is carried out, while the second rdtsc instruction records the counter value soon after the load instruction. The difference between the two counter values gives the duration of the load instruction. Since microarchitectural events last only for a few clock cycles, the contents of the eax register is sufficient. The higher bits of the counter stored in the edx register are generally not changed between the two rdtsc calls. The variable time would be higher if there is a cache miss compared to a cache hit.

Several sources of error are possible in the time measurements described above. A common source of error is from hardware interrupts that occur between the two rdtsc instructions. This would result in a measurement that is significantly larger than expected. Errors introduced by hardware interrupts can easily be eliminated by using a threshold. Values above the threshold would most probably be due to a hardware interrupt, therefore not considered. The value of the threshold should be set just above the maximum cycles required for a load.

Besides interrupts, several microarchitectural features could cause errors. These are subtle and more difficult to identify. The errors are considerably smaller, hence cannot be identified by thresholds. One such source of error is the out-of-order execution of instructions that is present in all x86 machines from Pentium Pro onwards. This feature allows instructions to be executed in an order that is different from that specified by the source code. For instance, since the load instruction in Listing 4.3 takes considerable amount of time, it may be performed earlier. This could potentially result in the first rdtsc instruction being performed after the load has completed. If this happened, the time measurement would not take the load into account.

In order to keep the rdtsc instruction from being performed out-of-order, a *serializing* instruction is required. This instruction waits until all preceding instructions complete. One such serializing instruction is the cpuid, which is normally used to identify the processor on which the program is executed. The cpuid instruction can be used to ensure that rdtsc is executed in-order. Listing 4.3 shows the time measurement using an rdtsc instruction that is serialized. Each rdtsc instruction is preceded and followed by a cpuid instruction. The proceeding cpuid instruction ensures that previous instructions are completed, while the cpuid instruction that follows ensures that no instruction is executed before the time-stamp counter is read for the second time. Newer x86 processors have an instruction rdtscp that reads the time-stamp counter and is a serialized instruction by itself.

Listing 4.3 *timestamp* (using rdtsc serialized rdtsc to measure the time required for a load instruction)

```
cpuid           ; ensure preceding instructions complete
rdtsc           ; read time stamp
cpuid           ; ensure preceding instructions complete
mov  time, eax  ; move counter into variable
load ebx, (ebp) ; a load from memory
cpuid           ; ensure preceding instructions complete
rdtsc           ; read time stamp again
cpuid           ; ensure preceding instructions complete
sub  eax, time  ; find the difference
```

Another source of error occurs in processors whose clock frequency is scaled depending on the work load. A processor that is lightly loaded runs at a lower clock frequency in order to save energy. Its clock frequency is increased automatically as the load of the processor increases in order to boost performance. This scaling of frequency could induce errors in the measured cycle count. It should, therefore, be ensured that the processor is sufficiently warmed up so that it runs at the maximum clock frequency before any measurements are made.

Systems can prevent the user space programs from using rdtsc and rdtscp instructions. However, high-precision timing measurements can still be made by using VTSCs. The next section discusses such counters.

4.2.2 Clocks with Virtual Time-Stamp Counters

To build a VTSC, two threads (A and B) are required, each running on a separate core of the processor and sharing a memory address CC, which mimics a time-stamp register. Thread A initializes one of the processor's general purpose register to 0, then increments it in an infinite loop, and copies the result to CC. The shared variable CC should be defined volatile to ensure timely copying. The loop would thus have an add instruction followed by a store. Assuming that no other process modifies CC, it would monotonically increment approximately once every clock cycle just like a hardware time-stamp counter. To use CC, thread B would read it before and after the instructions it wishes to measure.

Listing 4.4 shows an example of the VTSC to time an addition. The function `vtsc_thread` runs in a separate thread and increments the global volatile variable CC. This shared variable is read from the function `measure` before and after the addition is done. As in the case of rdtsc, the instruction cpuid may be added before and after CC is read to ensure serialized execution.

Listing 4.4 *timestamp* (using a virtualized time stamp counter to time an addition)

```
1
2  volatile unsigned long long CC;
3
4  void vtsc_thread(void *ptr)
5  {
6    while(1) CC++;
7  }
8
9  unsigned int measure(x, y)
10 {
11   t1 = CC;
12   z = x + y;
13   t2 = CC;
14
15   return (t2 - t1);
16 }
```

4.2.3 Distinguishing Cache Hit and Miss Events Using Time

Now that we have established how to measure time accurately, we look at how to use time to distinguish microarchitectural events such as a cache hit and a cache miss. We go back to Listing 4.1. The malicious user can invoke `AccessT` with chosen inputs and monitor the execution time required. A longer execution time would likely be due to a cache miss in the second access to the table T, while a shorter execution time would indicate a cache hit. To identify the cache hit, x_1 is fixed and different values of x_2 are chosen in each invocation of `AccessT`. The range of x_2 is 0 to 255. The values of x_2 that results in shorter execution time is likely to be due to a cache hit in the second access to table T.

However, things are not so straightforward. First, we first observe that in a system there are several microarchitectural components working simultaneously, each affecting execution time. This adds noise to the measurements. Since the difference between a cache hit and a miss is often just a few clock cycles, the noise would have a significant effect. The noise is due to several factors such as interrupts, direct memory access (DMA), instruction cache misses, activities in other processor cores, etc.

The noise can be reduced considerably by invoking `AccessT` several times with the same value of x_2 and then finding the average execution time. However, the noise is not uniform. For instance, a burst of network activity would momentarily add noise. This would affect some values of x_2 more than others. To avoid this, we randomize the order in which values of x_2 are chosen in the invocations of `AccessT`.

When a function is invoked repeatedly, the initial invocations take considerably more time. This is because some components in the system need to adjust to the new instructions. For instance, the RAM needs to be loaded with the new code,

the instruction cache needs to be filled with the new instructions, and the branch predictors needs to learn about the branches in the program. To avoid the initial overheads, we let the system have a warm up period by ignoring the first couple of invocations of `AccessT`. The warm up period also ensures that the processor frequency is scaled up to its maximum. Later invocations of `AccessT` would then be executed at the maximum frequency. Listing 4.5 shows how this is done.

Listing 4.5 Function FindK monitors execution time of `AccessT` to determine $\langle k_1 \oplus k_2 \rangle$

```
FindK(unsigned char x1, unsigned char x2){
  m = size of table in terms of number of blocks
  s = number of bits needed to address elements in a memory block
  T = {0};
  C = {0};
  x1 = 0×45;
  i = 0;

  while(i < 2^n){
    x2 = random() %256;
    flushcache();
    t1 = timestamp();
    AccessT(x1, x2);
    t2 = timestamp();

    i = i+ 1;
    if (i < 200) continue;                    /* the warm up */
    b = (x1 ⊕ x2) >>s;
    T[b] = T[b] + (t2− t1);
    C[b] = C[b] +1;
  }

  for (0 ≤ j < m){
    D[j] = (T[j]/C[j]);
  }

  return j for which D is minimum;
}
```

In the implementation

- m stores the number of memory blocks occupied by the look-up table. If the number of elements in the table is N, the size of each memory block is L , and size of each element in the table is B, then $m = (N \times B)/L$. For simplicity, we assume that the table is aligned to a block. In `AccessT`, for instance, there are 256 elements in table T, with each element occupying 1 byte. If we assume a memory block size of 64 bytes, then $m = 4$.
- s contains the number of bits needed to address elements in a memory block. This is equal to $\log_2(L/B)$. In the example configuration above $s = \log_2 64/1 = 6$.
- n determines the number of times the loop is run. As would be seen later in this chapter, as n increases, the probability of detecting the collision also increases.

Table 4.1 Average execution time for `AccessT` with $\langle x_1 \oplus x_2 \rangle$ in clock cycles on an Intel i7 Haswell Processor running at 3 GHz

$\langle x_1 \oplus x_2 \rangle$	Average execution time (in clock cycles)
0	484
1	483
2	446
3	457

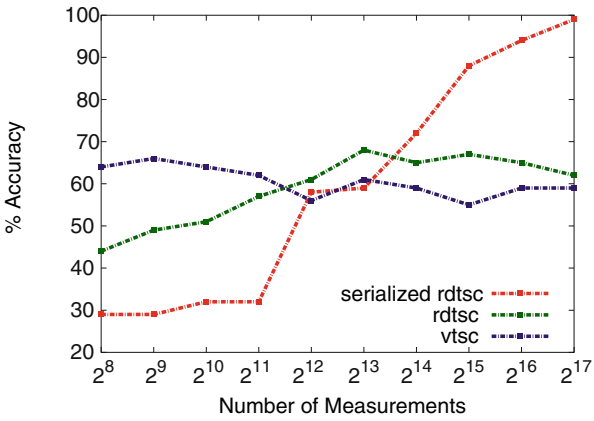


Fig. 4.3 Accuracy with which a cache hit is identified in `AccessT` with different clock sources. Large number of measurements along with a precise clock source can accurately identify cache hit events in the L1 data cache

- $\mathcal{T}$ is an array of size m , which accumulates the execution time for each block in the table.
- $\mathcal{C}$ is an array of size m , which counts the number of times a block in $\mathcal{T}$ is accessed in the second memory access.
- $\mathcal{D}$ computes the average execution time for each $\mathcal{T}$. The index of the block corresponding to the minimum $\mathcal{D}$ is returned.
- `flushcache` is a function that invalidates the contents of the look-up table in the cache memory.

If $k_1 \oplus k_2$ is 0×94, the two most significant bits is 2. Table 4.1 shows the average execution time for each of value of $\langle x_1 \oplus x_2 \rangle$. When $\langle x_1 \oplus x_2 \rangle = 2$, a cache hit occurs in the second access to the table T. All other values of $\langle x_1 \oplus x_2 \rangle$ results in a cache miss. As a result, the execution time for `AccessT` when $\langle x_1 \oplus x_2 \rangle = 2$ is slightly faster than the remaining. We, therefore, can infer that $\langle k_1 \oplus k_2 \rangle$ is 2.

The accuracy with which the cache hit is detected depends on several factors. The first is the penalty incurred by a cache miss. A small miss penalty would be more difficult to identify. The miss penalty depends on the system architecture and the level of the cache. Second, the precision with which timing measurements are made also critically affects the accuracy of a cache hit detection. The clock source should be precise enough to measure microvariations in execution time. The third factor affecting accuracy is the number of measurements that are considered while

computing the average execution time. A larger number of measurements would help eliminate noise to a greater extent. Figure 4.3 shows the accuracy with which the cache hit in `AccessT` is determined on an Intel i7 Haswell machine. As one would expect, the serialized rdtsc instruction provides the best accuracy when large number of measurements are made. With 2^{17} measurements for instance, cache hits are determined with an accuracy of about 99 %

4.3 Timing Attacks on Block Ciphers Based on Internal Collisions

Now that we have seen how time can be used to distinguish cache hits from misses, we show how it can be used to retrieve the secret key from a block cipher. We discuss this with AES (Advanced Encryption Standard) as an example. The 128-bit version of the AES cipher takes a plaintext block of 128 bits as input and a 128-bit secret key. Encryption starts with a key whitening followed by ten rounds. Each round except the last has four operations: SubBytes, ShiftRows, MixColumns, and AddRoundKeys. The last round does not have the MixColumns operation but has the three other operations. The SubBytes operation performs nonlinear transformations on the input, thereby adding confusion. The ShiftRows and MixColumns diffuse bits of the input. Details of the AES algorithm are present in Sect. 2.2.1.

Several implementations of AES are present. In the most common implementation, five 1024-byte look-up tables called T0, T1, T2, T3, T4 are used. The tables T0 to T3 are used in the first nine rounds. These tables abstract the SubBytes, ShiftRows, and MixColumns operations as shown in Sect. 2.2.1.1. Table T4 is exclusively used in the tenth round and abstracts the SubBytes and MixColumns operations. Listing 4.6 shows the first round of the code. This consists of 16 accesses to the tables.

Listing 4.6 First round of AES implementation

$$
\begin{aligned}
y_0 &= \mathcal{T}_0[s_0] \oplus \mathcal{T}_1[s_5] \oplus \mathcal{T}_2[s_{10}] \oplus \mathcal{T}_3[s_{15}] \oplus [k_0^{(1)}\ k_1^{(1)}\ k_2^{(1)}\ k_3^{(1)}]^T \\
y_1 &= \mathcal{T}_0[s_4] \oplus \mathcal{T}_1[s_9] \oplus \mathcal{T}_2[s_{14}] \oplus \mathcal{T}_3[s_3] \oplus [k_4^{(1)}\ k_5^{(1)}\ k_6^{(1)}\ k_7^{(1)}]^T \\
y_2 &= \mathcal{T}_0[s_8] \oplus \mathcal{T}_1[s_{13}] \oplus \mathcal{T}_2[s_2] \oplus \mathcal{T}_3[s_7] \oplus [k_8^{(1)}\ k_9^{(1)}\ k_{10}^{(1)}\ k_{11}^{(1)}]^T \\
y_3 &= \mathcal{T}_0[s_{12}] \oplus \mathcal{T}_1[s_1] \oplus \mathcal{T}_2[s_6] \oplus \mathcal{T}_3[s_{11}] \oplus [k_{12}^{(1)}\ k_{13}^{(1)}\ k_{14}^{(1)}\ k_{15}^{(1)}]^T \\
&\texttt{return } y\texttt{;}
\end{aligned}
$$

Algorithm 4.1: Strategy in a Time-Driven Attack for AES

Output: The value of $\langle k_a \oplus k_b \rangle$ for b ($b > a$)

1 **begin**
2 $d_a \leftarrow$ random value in the range $[0, 255]$.
3 **for** ρ *number of times* **do**
4 $\mathrm{x} \leftarrow (x_1 \| x_2 \| \cdots \| d_\mathrm{a} \| \cdots \| x_\mathrm{m})$ where $x_a = d_a$, $x_\mathrm{i} \xleftarrow{R} [0, 255]$, $1 \leq i \leq m$ and $i \neq a$.
5 $t \leftarrow$ Time(Encrypt using **x**).
6 Compute $\langle d_b \rangle$ from **x**.
7 $t_{\langle d_b \rangle} \leftarrow t_{\langle d_b \rangle} + t$
8 $c_{\langle d_b \rangle} \leftarrow c_{\langle d_b \rangle} + 1$
9 **end**
10 $\mathrm{t}_{avg\langle d_b \rangle} \leftarrow t_{\langle d_b \rangle} / c_{\langle d_b \rangle}$.
11 $\mathrm{t}_{avg} \leftarrow$ average encryption time for all encryptions.
12 **return** $\underset{\langle d_a \oplus d_b \rangle}{\mathrm{argmax}} (|\mathrm{t}_{avg\langle d_b \rangle} - \mathrm{t}_{avg}|)$.
13 **end**

In the first round, s_i has the form $x_i \oplus k_i$, where $1 \leq i \leq 16$. We see a similarity between Listings 4.1 and that of 4.6—both have loads from look-up tables that depend on the secret key. While the former has two key related table loads, the latter has 16; 4 to each look-up table. The time-driven attack in Listing 4.5 can therefore be extended to determine collisions between pairs of table accesses in AES. Algorithm 4.1 gives a high level overview.

To determine a collision between s_a and s_b ($1 \leq a < b \leq 16$) that access the table, the adversary keeps x_a constant, thereby fixing the table access made by s_a. For each of the memory blocks of the table accessible by $s_b = (x_b \oplus k_b)$, she determines the average execution time. Of all the average execution time computed, exactly one corresponds to a collision with s_a. This will have a distinctively different execution time compared to the others. If this value is identified, then we have found the collision. The collision reveals bits of $\langle k_a \oplus k_b \rangle$ as follows:

$$\begin{aligned} \langle s_a \rangle &= \langle s_b \rangle \\ \langle x_a \oplus k_a \rangle &= \langle x_b \oplus k_b \rangle \\ \langle k_a \oplus k_b \rangle &= \langle x_a \oplus x_b \rangle \end{aligned} \tag{4.4}$$

Algorithm 4.1 selects a value of d_a uniformly from the set 0 to 255 (line 2) and a plaintext **x** is chosen such that the value of d_a is fixed. The value of $\langle d_b \rangle$ is then computed from **x** and an encryption is invoked and timed. Lines 7 and 8 of the algorithm classifies the time. It comprises of a list of average timing corresponding to each line occupied by $\langle d_b \rangle$ (*i.e.,* the top $\log_2 l$ bits of d_b), Line 11 computes the overall average encryption time. The algorithm returns the value of $\langle d_a \oplus d_b \rangle$, which has

maximum deviation in time from the average case. An important requirement is that the remaining plaintext bytes (other than d_a and d_b) be sufficiently random in order to build the timing distribution. Algorithm 4.1 can be parallelized. That is for a fixed value of a, $\langle k_a \oplus k_b \rangle$ can be determined simultaneously for multiple values of b. In the parallel version multiple $t_{avg\langle d_b \rangle}$ are measured (one for each b). Consequently, the algorithm would return a set $\langle d_a \oplus d_b \rangle$.

Each look-up table in the AES implementation has 256 elements and each element occupies 4 bytes. On a system with a 64-byte memory block, the table would occupy 16 memory blocks with each memory block containing 16 elements. The timing side-channels essentially identifies a collision in a memory block. It would therefore reveal 4 bits. The keys k_a and k_b are bytes and have a combined entropy of 16 bits. The attack would reduce this entropy to $16 - 4 = 12$ bits.

Each look-up table used in the first round of AES is accessed four times. At most three collisions in each table can, therefore, be obtained. The AES implementation uses four look-up tables in the first round, therefore a maximum of 12 collisions are obtained. Each collision reveals 4 bits. In all 48 bits is revealed. Consequently, the entropy of the 128-bit AES key reduces to 80 bits.

The drawback of the attack is that it is highly sensitive to the initial state of the cache. A cache memory which is not cleaned at the start of encryption would either render the attack ineffective or adversely affect its efficiency. In section 4.4 we describe another timing attack that can reveal more bits about the keys and is not sensitive to the initial cache state.

4.3.1 Max, Min, or Max Deviation

Intuitively, a collision caused by a cache hit should result in a faster execution. Finding the collision using Algorithm 4.1 should have, therefore, identified the minimum average execution time. However in the algorithm, we chose to select the one with the maximum deviation from average. The reason for this is that in certain cases the collision could increase the execution time instead of reducing it.

This being said, the choice of maximum, minimum, or max deviation depends on the system and the cipher implementation. The attacker would try different choices to boost her success in obtaining the secret key. Chapter 6 provides the reason for this.

4.4 Time-Driven Attack Based on Induced Cache Miss

Suppose the function `AccessT` in Listing 4.1 were to be called twice with the same inputs and in quick succession. The first call may cause cache misses to occur due to the look-up table accesses. This would load part of the table into the cache. The second call to `AccessT` , on the other hand, would mostly incur cache hits, as a result is much faster. If a single block of the table is evicted from the cache in between invocations, then the second call to `AccessT` would incur a cache miss if the evicted block is accessed. However, a no cache miss is likely to occur if the evicted block is not accessed. The attack works as follows.

1. For inputs x_1 and x_2 , invoke `AccessT`.
2. Evict a cache line (say c) occupied by the table.
3. Invoke `AccessT` again with the same inputs and this time monitor the execution time.

A longer execution time would possibly be due to a cache miss that occurred. Since the invocations were made in quick succession, it is highly likely that the cache miss is due to the memory block at an offset c in the table being accessed. We thus know that at least one of the two accesses made to the table T were possibly to the memory block c. Therefore,

$$\begin{aligned} \langle c \rangle &= \langle x_1 \oplus k_1 \rangle \\ \textit{or} \\ \langle c \rangle &= \langle x_2 \oplus k_2 \rangle \end{aligned} \tag{4.5}$$

Thus,

$$\begin{aligned} \langle k_1 \rangle &= \langle x_1 \oplus c \rangle \\ \textit{or} \\ \langle k_1 \rangle &= \langle x_2 \oplus c \rangle \end{aligned} \tag{4.6}$$

We can determine which of these two equations is correct by changing the values of either x_1 or x_2 and checking if the cache miss persists. Listing 4.7 shows how it is done.

Listing 4.7 Function FindK monitors execution time of `AccessT` to determine $\langle k_1 \rangle$

```
FindK2(unsigned char x1, unsigned char x2){
  m = size of table in terms of number of blocks
  s = number of bits needed to address elements in a memory block
  T = {0};
  C = {0};
  i = 0;

  while(i < 2^n){
    x2 = random()%256;
    x1 = random()%256;
    line = random()%256;
    AccessT(x1, x2);          /* First Invocation */
    flushline(line);
    t1 = timestamp();
    AccessT(x1, x2);          /* Second Invocation */
    t2 = timestamp();

    i = i+1;
    if (i < 200) continue;              /* the warm up */

    k1 = (x1 ⊕ line) >> s;
    T[k1] = T[k1] + (t2 - t1);
    C[k1] = C[k1] +1;
  }

  for (0 ≤ j ≤ m){
    D[j] = (T[j]/C[j]);
  }

  return j for which D is minimum;
}
```

Details of the implementation are as follows:

- m stores the number of memory blocks occupied by the look-up table. For simplicity, we assume that the table is aligned to a block. In `AccessT` for instance, there are 256 elements in table $\mathcal{T}$. If we assume a memory block size of 64 bytes, then $m = 4$.
- s contains the number of bits needed to address elements in a memory block. In *AccessT*, each element in the table requires one byte. On a system with a 64-byte memory block $s = \log_2(64/1) = 6$.
- $\mathcal{T}$ is an array of size m, which accumulates the execution time for each block in the table.
- $\mathcal{C}$ is an array of size m, which counts the number of times a block in $\mathcal{T}$ is accessed in the second memory access.
- `flushline` is a function which invalidates the cache line that holds the element $T[line]$.

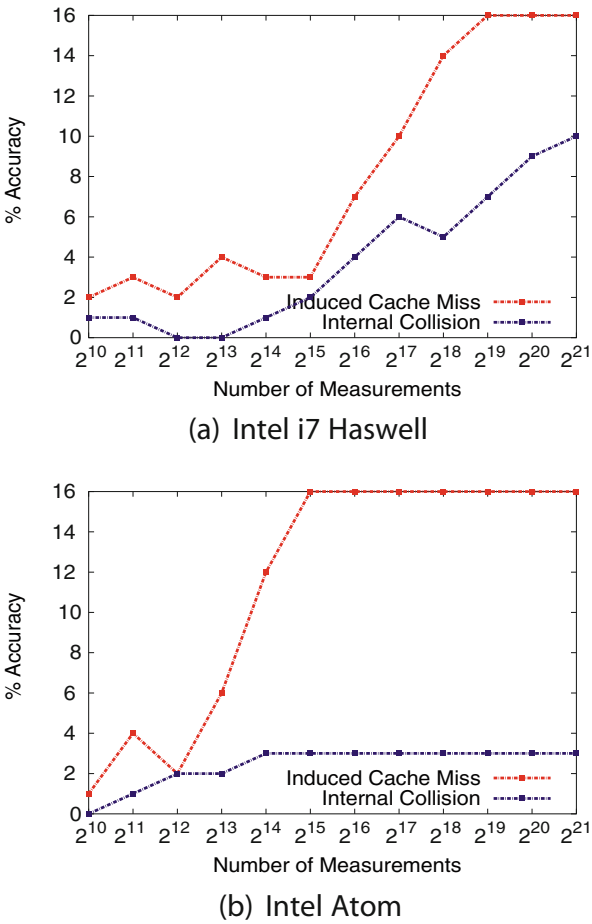


Fig. 4.4 Number of keys successfully determined in attacks on AES versus the number of measurements made. The induced cache miss technique is much more effective. As seen, it finds all keys correctly with significantly lesser number of timing measurements

The function `FuncK2` selects random values for x_1 and x_2 and invokes `AccessT` twice with these inputs. Another random value `line` is generated. This is used in the function `flushline` to evict a line from the cache, thereby potentially inducing a cache miss in the second invocation of `AccessT`. This invocation is timed and the loop is repeated several times. The value of $\langle k_1 \rangle$ is one that has the minimum execution time.

Working of `flushline`. The function `flushline` takes an input `line` and evicts T[`line`] from the cache if present. To be able to do this, `flushline` needs to know the cache sets which hold T. This is often not directly possible, since T is not accessible in function `FuncK2`. There are, however, indirect ways of obtaining the base address of T.

Consider that the adversary has an array as large as the cache memory. She first accesses all elements in the array. This would load the entire array into the cache.

She then invokes `AccessT` several times with different values of x_1 and x_2, hoping that the entire table T gets loaded into the cache. The accesses made to T would evict parts of the array from the cache. She then accesses the array again but this time times the accesses. The data in the array that take longer to load are possibly due to those evicted by T. She can thus identify the sets in the cache that hold T.

4.5 Results

Figure 4.4a shows the success in determining bits of the AES key on an Intel i7 Haswell system while Fig. 4.4b shows results on the Intel Atom system. The figures show that the induced cache miss is much better exploiting leakage compared to the using internal collisions.

4.6 Conclusion

Making accurate timing measurement is probably the most important step in a timing attack. The clocks used for making the measurements must be accurate enough to distinguish between microarchitectural events in the microprocessor, such as a cache hit and miss. This chapter described two ways to implement high precision clocks, namely using the in built *rdtsc* instruction and by software timestamp counters called VTSCs. These clock sources are used to analyze the memory access patterns of AES. The cache hits and misses identified from the AES execution is then used to develop two time-driven cache attacks that recover the AES secret key. Though the attacks work considerably well, the number of key bits they reveal is restricted by the size of the cache line. In the next chapter we discuss how cipher properties can be used to obtain more bits of the secret key.

Reference

1. Bertoni G, Zaccaria V, Breveglieri L, Monchiero M, Palermo G (2005) "AES Power Attack Based on Induced Cache Miss and Countermeasure," in ITCC (1). IEEE Computer Society, pp 586–591
2. Lauradoux C (2005) Collision attacks on processors with cache and countermeasures. In: Wolf C, Lucks S, Yau P-W (eds) WEWoRC, ser. LNI, vol 74. GI, pp 76–85
3. Fournier JJA, Tunstall M (2006) Cache based power analysis attacks on AES. In: Batten LM, Safavi-Naini R (eds) ACISP, ser. Lecture notes in computer science, vol 4058. Springer, pp 17–28
4. Bonneau J, Mironov I (2006) Cache-Collision timing attacks against AES. In: Goubin L, Matsui M (eds) CHES, ser. Lecture notes in computer science, vol 4249. Springer, pp 201–215
5. Aciiçmez O, Koç ÇK (2006) Trace-driven cache attacks on AES (Short Paper). In: Ning P, Qing S, Li N (eds) ICICS, ser. Lecture notes in computer science, vol 4307. Springer, pp 112–121

6. Page D (2002) Theoretical use of cache memory as a cryptanalytic side-channel. Departement of Computer Science, University of Bristol, Tech. Rep. http://eprint.iacr.org/2002/169
7. Tsunoo Y, Tsujihara E, Minematsu K, Miyauchi H (2002) Cryptanalysis of block ciphers implemented on computers with cache. In International Symposium on Information Theory and Its Applications, pp 803–806
8. Tsunoo Y, Saito T, Suzaki T, Shigeri M, Miyauchi H (2003) Cryptanalysis of DES Implemented on Computers with Cache. In: Walter CD, Kaya Koç Ç, Paar C (eds) CHES, ser. Lecture notes in computer science, vol 2779. Springer, pp 62–76
9. Aciiçmez O, Schindler W, Koç ÇK (2007) Cache based remote timing attack on the AES. In: Abe M (ed) CT-RSA, ser. Lecture notes in computer science, vol 4377. Springer, pp 271–286

Chapter 5
Advanced Time-Driven Cache Attacks on Block Ciphers

In Chapter 4 we saw how an adversary can use the execution time of the Advanced Encryption Standard (AES) block cipher to obtain information about the secret key. The number of key bits that the adversary determines is however restricted by the size of the cache line. If a cache line holds 2^δ elements of a look-up table used in AES, then at least δ bits of the key are hidden from the adversary. In this chapter, we discuss how properties of the block cipher along with timing side channels can be used to determine more key bits. The chapter begins with a second round time-driven cache attack on AES and then dwells into differential cache attacks on Feistel ciphers.

5.1 Second Round Attack on AES

The attack on AES discussed in Sect. 4.3 is able to determine relations of the form $\langle k_i^{(0)} \oplus k_j^{(0)} \rangle$ using collisions in the first round. The key bytes $k_i^{(0)}$ and $k_j^{(0)}$ $(0 \leq i, j < 15)$ are whitening key bytes. On a system with a 64-byte cache line, the entropy of the 128-bit AES key reduces to 80 bits (for details see Sect. 4.3). The second attack described in Section 4.4 performs better. It retrieves 4 bits of each of the 16 key bytes thus reducing the entropy of the AES key to $(128 - (4 \times 16)) = 64$ bits. In order to reduce the entropy further, Aciiçmez, Schindler, and Koç suggest targeting the second round of AES instead of the first [1]. Here we provide the basic second round attack technique. The attack considerably depends on the AES specification. Readers should refer Sect. 2.2.1 wherever needed.

Let a tilde over the symbol represent a guess. For instance, $\tilde{\mathbf{k}}_i^{(0)}$ is a guess for the whitening key byte $\mathbf{k}_i^{(0)}$, where $0 \leq i \leq 15$. If there is a collision between the states $s_0^{(0)}$ in the first round and $s_0^{(1)}$ in the second round (i.e., the first access to table $\mathcal{T}_0$ in the first and second rounds respectively), then Eq. 5.1 is obtained. In the equation y_0 is the index of the first access to the table $\mathcal{T}_0$ in the second round.

$$\begin{aligned} \langle s_0^{(0)} \rangle &= \langle s_0^{(1)} \rangle \\ \langle x_0 \oplus k_0^{(0)} \rangle &= \langle y_0 \oplus k_0^{(1)} \rangle \end{aligned} \tag{5.1}$$

C. Rebeiro et al., *Timing Channels in Cryptography,*
DOI 10.1007/978-3-319-12370-7_5

Using the AES algorithm (Sect. 2.2.1), y_0 can be written in terms of the plaintext and whitening key as follows.

$$y_0 = 2 \cdot S(x_0 \oplus \tilde{k}_0^{(0)}) \oplus 3 \cdot S(x_5 \oplus \tilde{k}_5^{(0)}) \oplus S(x_{10} \oplus \tilde{k}_{10}^{(0)}) \oplus S(x_{15} \oplus \tilde{k}_{15}^{(0)}) \quad (5.2)$$

Further, from the key expansion algorithm

$$k_0^{(1)} = k_0^{(0)} \oplus S(\tilde{k}_{13}^{(0)}) \oplus 1 \quad (5.3)$$

Substituting Eqs. 5.2 and 5.3 in Eq. 5.1, we obtain

$$\langle x_0 \rangle = \langle 2 \cdot S(x_0 \oplus \tilde{k}_0^{(0)}) \oplus 3 \cdot S(x_5 \oplus \tilde{k}_5^{(0)}) \oplus S(x_{10} \oplus \tilde{k}_{10}^{(0)}) \oplus S(x_{15} \oplus \tilde{k}_{15}^{(0)}) \oplus S(\tilde{k}_{13}^{(0)} \oplus 1) \rangle \quad (5.4)$$

There are five unknown key bytes involved: $\tilde{k}_0^{(0)}$, $\tilde{k}_5^{(0)}$, $\tilde{k}_{10}^{(0)}$, $\tilde{k}_{13}^{(0)}$, and $\tilde{k}_{15}^{(0)}$. These collectively take 2^{40} different values. For each of these values a set is created (thus there are 2^{40} sets). A random plaintext is chosen and the encryption time is found. This encryption time is added into all sets which satisfy Eq. 5.1 for the corresponding plaintext and key guess. The measurement is repeated several times with different plaintexts after which the average time in each of the 2^{40} key sets is found. The set in which the average time is the least corresponds to the correct key. Unlike the first round attacks, all bits of the five keys are obtained this way. In a same way other collisions between the first and second round can be used to determine the remaining key bytes.

The reason for the attack to work is as follows. The adversary chooses plaintexts randomly. Some of these plaintexts cause collisions between states $s_0^{(0)}$ and $s_0^{(1)}$ while others do not. Further, each plaintext will satisfy Eq. 5.1 for the correct key as well as for several wrong keys. Two cases arise,

- The execution time collected in a wrong key set may or may not correspond to collision between $s_0^{(0)}$ and $s_0^{(1)}$.
- The execution time in the set corresponding to the correct key always corresponds to collisions.

Thus, on average, the encryption time added to the correct set will have one cache miss less compared to the encryption time added to a wrong set. The smaller number of cache misses in the correct set will result in a smaller average execution time. This is used to distinguish the correct key.

5.2 Differential Cache Attacks on Feistel Ciphers

We now discuss attacks on Feistel ciphers that have a substitution–permutation (SP)-round function. These ciphers have a Feistel structure, with each round having a substitution followed by a permutation operation. This class of ciphers are of growing interest because they combine useful properties of SPN networks (like fast diffusion) with that of Feistel structures (like identical encryption and decryption). Notable

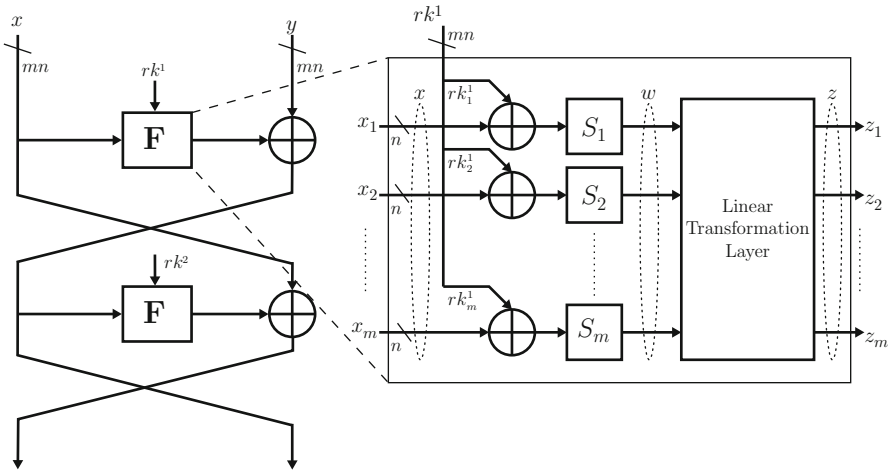


Fig. 5.1 Two rounds of a Feistel structure with SP-round function

block ciphers that have this structure are CAMELLIA [2], CLEFIA [3], and SMS4 [4]. The cipher E2 [5] is a deviant with an SPS round function (i.e., a permutation layer sandwiched between two substitution layers). To understand the working of the attack, we discuss with respect to a two-round Feistel structure having a SP-round function as shown in Fig. 5.1.

Let x and y be the arms of the Feistel structure with each arm having mn bits. The SP function, F, splits the input x into m equal parts (for example $x = (x_1|x_2|\cdots|x_m)$), each of n bits. The function F can be represented as

$$z = \mathsf{P}(\mathsf{S}_1(x_1 \oplus rk_1^1), \mathsf{S}_2(x_2 \oplus rk_2^1), \ldots, \mathsf{S}_m(x_m \oplus rk_m^1)) \,, \tag{5.5}$$

where S_1 to S_m are the substitution functions (s-boxes), P the permutation function that provides diffusion, and rk_i^1 (for $1 \leq i \leq m$) are n bit round key parts. The s-boxes in the F function are generally implemented as look-up tables in software. The number of tables used depends on the number of different s-boxes present in the cipher. Each table would have 2^n elements and occupy 2^l memory blocks (where $l = n - v$ and 2^v is the number of elements in a cache line).

Suppose the cache is initially clean and y is chosen such that all table accesses in the second round result in collisions, the following equality is obtained.

$$\langle x_i \oplus rk_i^1 \rangle = \langle z_i \oplus y_i \oplus rk_i^2 \rangle \qquad 1 \leq i \leq m \,, \tag{5.6}$$

where $\langle\,\rangle$ represents the most significant l bits, indicating the memory blocks occupied by the table.

Equation 5.6 can be deduced by monitoring the execution time using Algorithm 4.1. A cache hit in the first round can be determined by mapping $d_a \leftarrow x_i$ and $d_b \leftarrow x_j$ for $1 \leq i < j \leq m$ in the algorithm provided x_i and x_j access the same table. If there are g tables used in the F function, then there are a minimum of

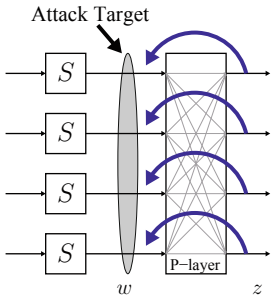


Fig. 5.2 The inverse of the permutation layer is used to obtain differences at the output of the s-box

g misses and a maximum of $m - g$ collisions that can be obtained in the first round (assuming a clean cache at the start of encryption). The state in which x results in the maximum collisions in the first round is known as the *one-round colliding state*.

A cache hit in the second round is obtained by choosing x so that a one-round colliding state is obtained and then mapping y_j $(1 \leq j \leq m)$ to d_b in Algorithm 4.1, while x_j is mapped to a corresponding d_a. Additionally x has to be kept constant. Thus Eq. 5.6 can be obtained with two invocations of Algorithm 4.1; the first invocation obtains collisions in the first round and the second invocation obtains collisions in the second round. In all 2ρ encryptions are required to be monitored—where ρ is the number of iterations of the loop in Algorithm 4.1.

Now consider another input $x' = (x'_1|x'_2|\cdots x'_m)$ such that $x' \neq x$. Let x' result in the s-box output z', and y' the corresponding input which causes collisions in the second round. These collisions would result in the following equalities:

$$\langle x'_i \oplus k^1_i \rangle = \langle z'_i \oplus y'_i \oplus rk^2_i \rangle \quad \text{where } 1 \leq i \leq m. \tag{5.7}$$

Adding Eqs. (5.6) and (5.7) we obtain

$$\langle x_i \oplus x'_i \oplus y_i \oplus y'_i \rangle = \langle z_i \oplus z'_i \rangle. \tag{5.8}$$

Thus the differences in the inputs $\Delta x_i = x_i \oplus x'_i$ and $\Delta y_i = y_i \oplus y'_i$ can be used to determine the output difference of the F function in the first round.

For Eq. 5.7, ρ encryptions are sufficient (or 1 invocation of Algorithm 4.1 because the collisions in the first round can be established for free using the previously determined equalities $\langle rk^1_i \oplus rk^1_j \rangle = \langle x_i \oplus x_j \rangle$, $1 \leq i < j \leq m$. Thus in total, 3ρ encryptions are needed to obtain Eq. 5.8.

In order to derive candidates for the round key rk^1, we use the inverse of the permutation layer (Fig. 5.2), which can be represented as m functions having the form $w_j = LT^{-1}_j(z)$, where $1 \leq j \leq m$. For the input difference Δx, the collisions in the second round would reveal $\langle \Delta z_i \rangle$ (for all $1 \leq i \leq m$). These can be used in $LT^{-1}_j(\cdot)$ to obtain few bits of Δw_j (for all $1 \leq j \leq m$); the output difference of the s-box. The input difference and the partial output difference of the s-box is then used to reduce the key space of rk^1_j.

Expected Number of Candidate Keys For any of the $n \times n$ s-boxes in Fig. 5.1, the total number of output differences for a given input difference is 2^{n-1}. Let o_b be the number of bits of Δw_j that gets revealed by the application of $LT_j^{-1}(\cdot)$. The number of bits o_b can have a minimum value of 0 and a maximum of l. The possible number of output differences of the s-box gets reduced to 2^{n-o_b-1}. This reduces the set of candidate keys for k_j^1 from 2^n to N_b. On average N_b is

$$N_b = 2^{(n-o_b)} . \tag{5.9}$$

5.3 Differential Cache Attack on CLEFIA

Now that we have developed the theory behind differential cache attacks, we show how it can be used on the block cipher CLEFIA.

CLEFIA is a 128-bit block cipher and a standard for lightweight encryption. It has a double Feistel structure and uses two F functions named $F0$ and $F1$ (Sect. 2.2.2). Analysis of the CLEFIA algorithm shows that the knowledge of any set of four round keys ($RK4i, RK4i + 1, RK4i + 2, RK4i + 3$), where $i \bmod 2 = 0$, is sufficient to revert CLEFIA's key expansion process and obtain the secret key. In the differential cache attack on CLEFIA that we describe here, round keys $RK0$, $RK1$, $RK2$, and $RK3$ are determined from which K is obtained.

The attack on CLEFIA comprises three steps. First $RK0$ and $RK1$ are determined, then $WK0 \oplus RK2$ and $WK1 \oplus RK3$, and finally $RK4$ and $RK5$. With these round keys, CLEFIA's key expansion algorithm is used to obtain 57 bits of ($RK2|RK3$). In all, obtaining the 121 bits of the round keys $RK0$, $RK1$, $RK2$, and $RK3$ requires 2^{14} collisions.

5.3.1 Differential Properties of CLEFIA's F Functions

The attack uses the following observations on the F functions:

- Matrices $M0$ and $M1$ in the F functions do not attain complete diffusion in all bits of the output. If the five most significant bits (*MSBs*) of the input of each byte of the matrices $M0$ and $M1$ are known then few bits of the output can be computed (see Fig. 2.4). In particular three *MSBs* of each byte in $M0$'s output and two *MSBs* of each byte in $M1$'s output are computable. Since $M0$ and $M1$ are self inverting matrices, the inverse of the above statement also holds. That is, given five *MSBs* of each byte of the output, three *MSBs* of the input in $M0$ and two *MSBs* of the input in $M1$ is computable.
- For a pair of inputs, the non-linearity in the s-boxes causes several (60 % in $S0$ and 50 % in $S1$) of the possible input difference–output difference combinations to be invalid. Additionally, for a valid combination, $S0$ has 1.28 choices on average for the inputs to the s-box, while $S1$ has 1.007.

If the inputs to an s-box is $(x_i \oplus k)$ and $(x'_i \oplus k)$, then the ex-or difference is $(x_i \oplus x'_i)$. This is known to the adversary. Additionally, on a system with a 32-byte cache line, the cache traces reveal three bit differences per byte of the output of each s-box of $F0$. For the remaining five bits of each output, there are 32 possible input difference–output differences for each s-box resulting in an average of 32 key (k) candidates for each byte. Similarly there are about 64 choices for each key byte in $F1$. We now show how these differential properties of CLEFIA are used in the recovery of the round keys.

5.3.2 *Determining* $RK0$ *and* $RK1$

The equations for the indices to the tables in the first round is given by:

$$\begin{aligned} I1^0_{s0} &= P_0 \oplus RK0_0 & I1^1_{s0} &= P_2 \oplus RK0_2 \\ I1^2_{s0} &= P_9 \oplus RK1_1 & I1^3_{s0} &= P_{11} \oplus RK1_3 \\ I1^0_{s1} &= P_1 \oplus RK0_1 & I1^1_{s1} &= P_3 \oplus RK0_3 \\ I1^2_{s1} &= P_8 \oplus RK1_0 & I1^3_{s1} &= P_{10} \oplus RK1_2 \end{aligned} \tag{5.10}$$

where $I\alpha^i_{s\beta}$ denotes the index to the $(i+1)$th access to table $s\beta$ in round α.

If we make the assumption that no part of the table is present in cache before the start of encryption, then the first access to each table, i.e., $I1^0_{s0}$ and $I1^0_{s1}$, results in cache misses. Keeping P_0 and P_1 fixed and by varying the other bytes in Algorithm 4.1, collisions can be determined for all table accesses in the first round. Such a state of the cipher is called a *one-round colliding state*.

In the second round, the indices to the tables $S0$ and $S1$ in $F0$ are given by equations in (5.11), where $P_{(0\cdots3)}$ indicates the concatenation of P_0, P_1, P_2, and P_3.

$$\begin{aligned} I2^0_{s0} &= P_4 \oplus WK0_0 \oplus F0(RK0, P_{(0\cdots3)})_0 \oplus RK2_0 \\ I2^0_{s1} &= P_5 \oplus WK0_1 \oplus F0(RK0, P_{(0\cdots3)})_1 \oplus RK2_1 \\ I2^1_{s0} &= P_6 \oplus WK0_2 \oplus F0(RK0, P_{(0\cdots3)})_2 \oplus RK2_2 \\ I2^1_{s1} &= P_7 \oplus WK0_3 \oplus F0(RK0, P_{(0\cdots3)})_3 \oplus RK2_3 \end{aligned} \tag{5.11}$$

Starting from the one-round colliding state, four collisions are forced in $F0$ of the second round by varying, P_4, P_5, P_6, and P_7. This results in identifying five collisions in table $S0$ (three in the first round and two in the second). The *MSBs* of the indices to the table are all the same, *i.e.*, $\langle I1^0_{s0}\rangle = \langle I1^1_{s0}\rangle = \langle I2^0_{s0}\rangle = \langle I2^1_{s0}\rangle$. We therefore get the following equalities:

$$\langle P_0 \oplus P_4\rangle = \langle F0(RK0, P_{(0\cdots3)})_0 \oplus RK0_0 \oplus WK0_0 \oplus RK2_0\rangle \tag{5.12}$$

$$\langle P_2 \oplus P_6 \rangle = \langle F0(RK0, P_{(0\cdots3)})_2 \oplus RK0_2 \oplus WK0_2 \oplus RK2_2 \rangle$$

Similarly the five cache hits in table $S1$ result in the following equalities:

$$\begin{aligned} \langle P_1 \oplus P_5 \rangle &= \langle F0(RK0, P_{(0\cdots3)})_1 \oplus RK0_1 \oplus WK0_1 \oplus RK2_1 \rangle \\ \langle P_3 \oplus P_7 \rangle &= \langle F0(RK0, P_{(0\cdots3)})_3 \oplus RK0_3 \oplus WK0_3 \oplus RK2_3 \rangle \end{aligned} \tag{5.13}$$

For another plaintext Q, with $Q_0 \neq P_0$ and $Q_1 \neq P_1$, equations similar to (5.12) and (5.13) can be obtained by tracing cache collisions in the first and second rounds. These are shown in Eq. 5.14, where $0 \leq i < 4$.

$$\langle Q_i \oplus Q_{4+i} \rangle = \langle F0(RK0, Q_{(0\cdots3)})_i \oplus RK0_i \oplus WK0_i \oplus RK2_i \rangle \tag{5.14}$$

From Eqs. 5.12, 5.13, and 5.14, and the fact that $\langle P_0 \oplus P_2 \oplus P_4 \oplus P_6 \rangle = \langle Q_0 \oplus Q_2 \oplus Q_4 \oplus Q_6 \rangle$, and $\langle P_1 \oplus P_3 \oplus P_5 \oplus P_7 \rangle = \langle Q_1 \oplus Q_3 \oplus Q_5 \oplus Q_7 \rangle$ the following equations are generated:

$$\begin{aligned} \langle P_0 \oplus P_4 \oplus Q_0 \oplus Q_4 \rangle &= \langle F0(RK0, P_{(0\cdots3)})_0 \oplus F0(RK0, Q_{(0\cdots3)})_0 \rangle \\ \langle P_1 \oplus P_5 \oplus Q_1 \oplus Q_5 \rangle &= \langle F0(RK0, P_{(0\cdots3)})_1 \oplus F0(RK0, Q_{(0\cdots3)})_1 \rangle \\ \langle P_2 \oplus P_6 \oplus Q_2 \oplus Q_6 \rangle &= \langle F0(RK0, P_{(0\cdots3)})_2 \oplus F0(RK0, Q_{(0\cdots3)})_2 \rangle \\ \langle P_3 \oplus P_7 \oplus Q_3 \oplus Q_7 \rangle &= \langle F0(RK0, P_{(0\cdots3)})_3 \oplus F0(RK0, Q_{(0\cdots3)})_3 \rangle \end{aligned} \tag{5.15}$$

It is now possible to apply the differential properties of the F functions to derive possible key candidates. Considering just two blocks of plaintexts, P and Q, would result in 32 candidate key values (on average) for each byte of $RK0$. In order to identify a single key with probability greater than $1/2$, cache hits in four plaintexts must be considered, and the intersection between all possible candidate key sets must be found.

In a similar way round key $RK1$ can be determined by analyzing cache hits in $F1$. The set of equations that should satisfy $RK1$ is shown below, where $0 \leq i < 4$.

$$\langle P_{8+i} \oplus P_{12+i} \oplus Q_{8+i} \oplus Q_{12+i} \rangle = \langle F1(RK0, P_{(8\cdots11)})_i \oplus F1(RK0, Q_{(8\cdots11)})_i \rangle$$

Due to the matrix $M1$, which only reveals two bits of the difference of outputs in each s-box, six plaintext blocks are required instead of four.

5.3.3 *Determining* $WK0 \oplus RK2$ *and* $WK1 \oplus RK3$

A cache hit in the first table accesses in the third round can be found by varying byte P_8 for $S0$ (and P_9 for $S1$) (Fig. 2.4). The cause of this cache hit could be collisions with any of the eight previous accesses to that table. To reduce the number of "causes" that result in cache hits, the plaintext bytes are chosen in a way such that the first two rounds have only one cache miss in each table (i.e., the first accesses).

Such a state of the cipher is called the *two-round colliding state*. The two-round colliding state has 14 cache hits in the first two rounds. Such a state is obtained by first obtaining the one-round colliding state and then varying bytes P_4 to P_7 and P_{12} to P_{15} independently until eight cache hits in the second round are also obtained.

The third round first access cache hit caused by changing P_8 (or P_9) starting from the two-round colliding state has three causes. The first two causes are due to collisions with $S0$ table accesses in $F1$ in round two. The third cause is due to collisions with $S0$ accesses in $F0$; this is of interest and is estimated to occur once every three collisions. The uninteresting cache hits due to the first two reasons are caused by the changing P_8, which in turn changes $Y1\text{-}1$ (Fig. 2.4). On obtaining a cache hit in the first table access in the third round, it is required to identify whether the hit is interesting. This is done by changing the value of P_{12} (or P_{13}) and re-doing the encryption. If a cache hit still occurs in round three, then with significant probability it is of interest.

Similar cache hits for the other $F0$ table accesses in round three can be obtained. With these collisions the following equalities are satisfied for a pair of plaintexts P and Q.

$$\begin{aligned}\langle P_i \oplus Q_i \oplus P_{8+i} \oplus Q_{8+i} \rangle = &\langle F0(RK2, WK0 \oplus P_{(4\cdots7)} \oplus Y0\text{-}1^P)_i \\ &\oplus F0(RK2, WK0 \oplus Q_{(4\cdots7)} \oplus Y0\text{-}1^Q)_i \rangle,\end{aligned}$$

where $0 \leq i < 4$, and $Y0\text{-}1$ is as defined in Fig. 2.4. $Y0\text{-}1$ can be computed using the $RK0$ found in the first step of the attack. Differential properties of the $F0$ function and four plaintext blocks can be used to completely determine $RK2 \oplus WK0$.

Finding WK1 ⊕ RK3 In a similar manner $RK3 \oplus WK1$ can be found in less than 2^{12} invocations of Algorithm 4.1 by considering collisions in $F1$ in round three and varying plaintext bytes $P_{(0\ldots3)}$. The difference equations that is to be satisfied is given by the following (where $0 \leq i < 4$) :

$$\begin{aligned}\langle P_i \oplus Q_i \oplus P_{8+i} \oplus Q_{8+i} \rangle = &\langle F1(RK3, WK1 \oplus P_{(12\cdots15)} \oplus Y1\text{-}1^P)_i \\ &\oplus F1(RK3, WK1 \oplus Q_{(12\cdots15)} \oplus Y1\text{-}1^Q)_i \rangle\end{aligned}$$

5.3.4 Determining $RK4$ and $RK5$

$RK4$ and $RK5$ can be determined in 2^{13} invocations of Algorithm 4.1 using the same idea as the second step of the attack. To find $RK4$, a two-round colliding state is first obtained from which cache hits in $F0$ of the fourth round is forced by varying the fourth word of the plaintext. $RK4$ can be determined from this using the equations:

$$\begin{aligned}&\langle P_i \oplus Q_i \oplus P_{12+i} \oplus Q_{12+i} \oplus Y1\text{-}1_i^P \oplus Y1\text{-}1_i^Q \rangle \\ &\qquad = \langle F0(RK4, X0\text{-}3^P)_i \oplus F0(RK4, X0\text{-}3^Q)_i \rangle,\end{aligned}$$

where $Y1\text{-}1^P$, $Y1\text{-}1^Q$, $X0\text{-}3^P$, and $X0\text{-}3^Q$ are computed from previously determined round keys and $0 \leq i < 4$. Similarly, $RK5$ is determined by cache hits in $F1$ in the 4th round. The equalities for determining $RK5$ are :

$$\langle P_{4+i} \oplus Q_{4+i} \oplus P_{8+i} \oplus Q_{8+i} \oplus Y0\text{-}1_i^P \oplus Y0\text{-}1_i^Q \rangle = \langle F1(RK5, X1\text{-}3^P)_i \oplus F1(RK5, X1\text{-}3^Q)_i \rangle$$

5.3.5 *Determining* $RK2$ *and* $RK3$

In the key expansion algorithm, if $i = 0$ then $T = (RK0|RK1|RK2|RK3)$, and $T = L \oplus (CON_{24}|CON_{25}|CON_{26}|CON_{27})$. Sixty-four bits of the key dependent constant L can be computed using the values of $RK0$ and $RK1$, which were determined in the first step of the attack.

$$(L_0|L_1) = (RK0|RK1) \oplus (CON_{24}|CON_{25}) \tag{5.16}$$

The double swap operation on L places 57 known bits of L in the lower bit positions. This is given by $L'_{(0\cdots56)} = L_{(7\cdots63)}$. Again, in the key expansion algorithm, if $i = 1$, then $T = (RK4|RK5|RK6|RK7)$. This is represented as $T = L' \oplus (CON_{28}|CON_{29}|CON_{30}|CON_{31}) \oplus (WK0|WK1|WK2|WK3)$. Therefore,

$$WK0|WK1_{(0\cdots24)} = L'_{(0\cdots56)} \oplus (CON_{28}|CON_{29(0\cdots24)}) \oplus (RK4|RK5) \tag{5.17}$$

Using $RK4$ and $RK5$ that were determined in the third step of the attack, the whole of $WK0$ and 25 bits of $WK1$ can be determined. Then the result from the second step of the attack is used to obtain 57 bits of $(RK2|RK3)$. Thus 121 out of the 128 bits of $(RK0|RK1|RK2|RK3)$ is retrieved.

5.4 Conclusion

Using properties of block ciphers along with information leakage from timing channels lead to deadly attacks. This chapter showed how properties of AES and CLEFIA can be used to build efficient time-driven cache attacks. The attack on AES for instance requires just 2^{16} time measurements to reveal all bits of the secret key. These attacks must be prevented with as little overheads as possible and with guarantees of security. To do so, we would require an in depth analysis of the attack and the leakage in the timing channels. The next chapter builds a framework to study the information leakage from cache memories. The framework is used to build efficient implementations of block ciphers that are secure against this class of timing attacks.

References

1. Aciiçmez O, Schindler W, Kaya Koç Ç (2007) Cache based remote timing attack on the AES. In: Abe M (ed) CT-RSA. Lecture notes in computer science, vol 4377. Springer, Berlin, pp 271–286
2. Aoki K, Ichikawa T, Kanda M, Matsui M, Moriai S, Nakajima J, Tokita T (2000) Camellia: a 128-bit block cipher suitable for multiple platforms—design and analysis. In: Stinson DR, Tavares SE (eds) Selected areas in cryptography. Lecture notes in computer science, vol 2012. Springer, Berlin, pp 39–56
3. Shirai T, Shibutani K, Akishita T, Moriai S, Iwata T (2007) The 128-bit blockcipher CLEFIA (extended abstract). In: Biryukov A (ed) FSE. Lecture notes in computer science, vol 4593. Springer, Berlin, pp 181–195
4. Diffie W, Liden G (trans) (2008) SMS4 encryption algorithm for wireless networks. Cryptology ePrint Archive, Report 2008/329 (http://eprint.iacr.org/)
5. Kanda M, Moriai S, Aoki K, Ueda H, Takashima Y, Ohta K, Matsumoto T (2000) E2—a new 128-bit block cipher. IEICE Trans Fundam Electron Commun Comput Sci E82(A-1):48–59
6. Fournier JJA, Tunstall M (2006) Cache based power analysis attacks on AES. In: Batten LM, Safavi-Naini R (eds) ACISP, ser. Lecture notes in computer science, vol 4058. Springer, pp 17–28
7. Gallais JF, Kizhvatov I, Tunstall M (2010) Improved trace-driven cache-collision attacks against embedded AES implementations. In: Chung Y, Yung M (eds) WISA, ser. Lecture notes in computer science, vol 6513. pp 243–257
8. Rebeiro C, Poddar R, Datta A, Mukhopadhyay D (2011) An enhanced differential cache attack on CLEFIA for large cache lines. In the Proceedings of the 12th International Conference on Cryptology in India, INDOCRYPT 2011, Chennai, India, pp 58–75, LNCS 7107
9. Poddar R, Datta A, Rebeiro C (2011) A cache trace attack on CAMELLIA. In the Proceedings of First International Conference on Security Aspects in Information Technology, InfoSecHiComNet 2011, Haldia, India, pp 141–156, LNCS 7011
10. Rebeiro C, Mukhopadhyay D (2011) Cryptanalysis of CLEFIA using differential methods with cache trace patterns. In the Proceedings of the Topics in Cryptology—CT-RSA 2011—The Cryptographers' Track at the RSA Conference 2011, San Francisco, CA, USA, pp 89–103, LNCS
11. Rebeiro C, Nguyen PH, Mukhopadhyay D, Pochmann A (2013) Formalizing the effect of feistel cipher structures on differential cache attacks. IEEE Trans Inf Foresics Secu 8(8):1274–1279
12. Nguyen PH, Rebeiro C, Mukhopadhyay D, Huaxiong W (2012) Improved differential cache trace attacks on SMS4. In the Proceedings of the 8th International Conference, Inscrypt 2012, Beijing, China, pp 29–45, LNCS 7763

Chapter 6
A Formal Analysis of Time-Driven Cache Attacks

Chapter 4 introduced two forms of time-driven cache attacks. It showed how the system configuration, number of measurements taken, and the attack algorithm can adversely affect an adversary's success. In this chapter we build a formal framework to analyze time-driven cache attacks on block ciphers. The framework is used to determine the causes of information leakage in modern cache memories. The framework shows that in these memories, there are two causes of information leakage. The first is due to the number of cache misses that occur during the encryption while the second is due to microarchitectural features in the cache such as nonblocking, pipelined, out-of-order, and parallel servicing of cache misses. The framework is used to evaluate commonly used block ciphers and identify the best way to implement them.

6.1 Memory Access Model for a Block Cipher

A block cipher takes as input a plaintext block (typically 64 or 128 bits) and encrypts it with a secret key to produce a ciphertext. Typically block ciphers first subject the plaintext to a key whitening phase where a secret key is added. These are the keys a side-channel adversary tries to recover. The whitening phase is followed by a number of iterations called rounds. Each round has three operations: key mixing, substitution, and permutation. Key mixing modifies the intermediate state of the cipher by adding secret data to it. Permutation permutes bits in the state, while substitution adds randomness to the state. Figure 6.1 shows a typical block cipher structure.

When speed of encryption is important, the bottleneck comes from the substitution operation. To improve speeds, developers implement the substitution operations using look-up tables, denoted $\mathcal{T}$ in Fig. 6.1. In some block cipher implementations, further speed-ups are achieved by huge tables that implement both the substitution and permutation. These tables are known as T-tables. For instance, see the implementation of Advanced Encryption Standard (AES) in Sect. 2.2.1.1 and CLEFIA in Sect. 2.2.2.1 use T-tables.

C. Rebeiro et al., *Timing Channels in Cryptography,*
DOI 10.1007/978-3-319-12370-7_6

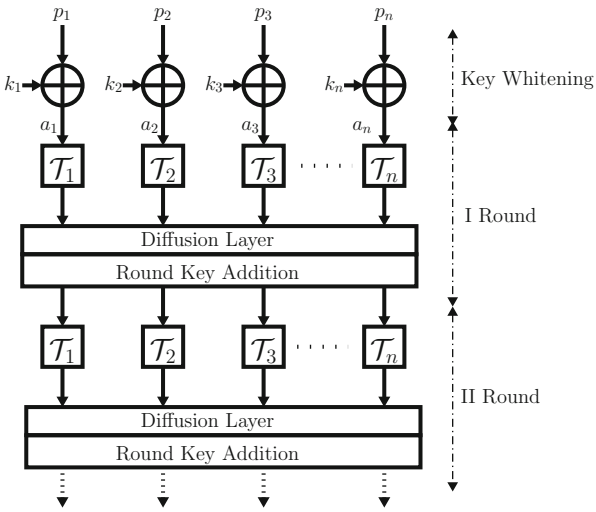


Fig. 6.1 Typical iterative block cipher structure. Each *round* comprises functions that perform key mixing, substitution, and permutation

Most block cipher implementations can be mapped to the structure shown in Fig. 6.1 using five parameters. Table 6.1 shows these parameters while Table 6.2 shows some examples of block cipher implementations mapped to this structure. In later sections we use this model of the block cipher to analyze time-driven cache attacks.

6.2 Cache Misses in a Block Cipher

A block cipher execution makes several memory accesses during its execution due to the load and store instructions. However, when cache attacks are considered, we are only interested in the memory loads or stores of key related data—for instance when the memory location accessed depends on the key. The cache hits or misses that occurs as a result leaks information about the secret key. In block ciphers such as in Fig. 6.1, such key related memory accesses occur due to the loads from the look-up tables. In this section we mathematically analyze the cache misses that occur due to such memory loads.

Suppose the cipher's memory model (Table 6.1) has g look-up tables denoted $\mathcal{T}_1, \mathcal{T}_2, \ldots, \mathcal{T}_g$, each occupying l memory blocks and accessed m times per round in γ rounds. We define the set $\mathbb{S}_i$ on the look-up table $\mathcal{T}_i$ (where $1 \leq i \leq g$) as follows:

Definition 6.1 Let $\mathbb{S}_i$ be a set of sequences defined as $\mathbb{S}_i = \{(p_{i,1}, p_{i,2}, \ldots, p_{i,\gamma}) \mid$ where $p_{i,r}$ is the number of cache misses due to $\mathcal{T}_i$ that occur in the rth round of an encryption and $1 \leq r \leq \gamma\}$.

Table 6.1 Memory access parameters of a block cipher

γ	Number of rounds in the cipher. Each round has a layer of substitution, permutation, and key mixing
g	Number of look-up tables used in the implementation. This includes the look-up tables implemented as T-tables
m	Number of key related look-ups per look-up table per round
l	Number of memory blocks required to hold a table (*table size*). We assume that all look-up tables are of the same size
2^{δ}	Number of elements in the table sharing a memory block

Table 6.2 Memory access parameters of popular block cipher implementations

	g	γ	**m**	**l**	$\mathbf{2^{\delta}}$	T-tables
CLEFIA[a]	2	18	4	4	64	No
CAMELLIA[b]	4	18	2	4	64	No
DES[c]	8	16	1	4	16	Yes
AES[d]	4	9	4	16	16	Yes

DES Data Encryption Standard, *AES* Advanced Encryption Standard
[a] Reference Code 1.0.0, *http://www.sony.net/Products/cryptography/clefia*
[b] PolarSSL 1.1.1, *http://polarssl.org/*
[c] OpenSSL 0.9.8a, *http://www.openssl.org*. The last round is not considered in the model because it uses a different table. However all empirical results reported in this work include the last round as well
[d] In all cases we assume a cache memory with a cache line size of 64 bytes. Therefore, in bytes the look-up table has size $64 \cdot l$ and number of elements $2^{\delta} \cdot l$

For instance, a possible element in $\mathbb{S}_i$ is $\{4, 3, 4, 1, 0, \ldots, 0\}$. This means that due to table $\mathcal{T}_i$ there were four cache misses in the first round, three in the second, four in the third round, and so on.

The number of cache misses in the m accesses to look-up table $\mathcal{T}_i$ in each round of the cipher are discrete random variables. We denote them by $(\mathsf{P}^m_{i,1}, \mathsf{P}^m_{i,2}, \ldots, \mathsf{P}^m_{i,\gamma})$. Any sequence in $\mathbb{S}_i$ satisfies the following properties:

- If we assume that the contents of the cache is flushed before encryption then at least one cache miss due to $\mathcal{T}_i$ will occur in the first round. Further if we assume no conflict misses during encryption, then a memory block once loaded into the cache would remain in the cache till the encryption completes.
  With these assumptions, the minimum number of cache misses that occur in the first round due to table $\mathcal{T}_i$ is 1. In this case, the first load from the table results in a compulsory miss. All other accesses to $\mathcal{T}_i$ in the round are cache hits. The lower bound for $\mathsf{P}^m_{i,1}$ is therefore 1.
- In rounds other than the first, all accesses to table $\mathcal{T}_i$ may result in cache hits. Thus the minimum number of cache misses in the round is 0. In other words, the lower bound for $\mathsf{P}^m_{i,r}$ $(2 \leq r \leq \gamma)$ is 0.

- The number of cache misses in the round due to $\mathcal{T}_i$ is limited by its size (i.e., l) and the number of accesses made to the table in the round (m): $\mathsf{P}^m_{i,r} \leq minimum(l, m)$ $(1 \leq r \leq \gamma)$.
- The sum of all elements in the sequence is limited by l—the size of $\mathcal{T}_i$. That is, $\sum_{r=1}^{\gamma} p_{i,r} \leq l$.

The following theorem computes the probability of obtaining $p_{i,r}$ cache misses in the rth round for table $\mathcal{T}_i$.

Theorem 6.1 *Let* $\mathrm{Q}_{i,r}$ *be a discrete random variable denoting the number of cache misses due to table* $\mathcal{T}_i$ *that have occurred before the start of round* r *(i.e., in rounds* 1 *to* $r-1$*). Three cases arise:*

- *When* $r = 1$ *then* $\mathrm{Q}_{i,1} = 0$ *and* $\Pr[\mathrm{P}^m_{i,1} = 0] = 0$.
- *When* $r > 1$ *then*

$$\Pr[\mathrm{P}^m_{i,r} = 0 | \mathrm{Q}_{i,r} = q_{i,r}] = \left(\frac{q_{i,r}}{l}\right)^m .$$

- *Otherwise,*

$$\Pr[\mathrm{P}^m_{i,r} = p | \mathrm{Q}_{i,r} = q] = \frac{1}{l^m} \binom{l-q}{p} \sum_{j=1}^{p} (-1)^{p-j} \binom{p}{j} \left[(j+q)^m - q^m\right] .$$

(In this above equation we have denoted $p_{i,r}$ *by* p *and* $q_{i,r}$ *by* q *for simplicity of notation.)*

where $q = q_{i,r} = p_{i,1} + p_{i,2} \cdots + p_{i,r-1}$ *and* $q_{i,1} = 0$*; and* $p = p_{i,r}$ *can take values from* 0 *to the minimum of* m *and* $l - q_{i,r}$.

Proof. Since we assume that the cache is flushed before starting the encryption, in the first round ($r = 1$), $\mathsf{P}^m_{i,1} = 0$ is not possible because at least one compulsory cache miss will occur.

Assuming no conflicts, when $r > 1$, no cache misses will occur if the table accesses were made to blocks that were previously accessed, so data would be read from the cache. Each of the m accesses in the round are therefore cache hits and can take $q_{i,r}$ possible values.

The third case determines the probability of obtaining p cache misses in the rth round given that the previous $r-1$ rounds resulted in q cache misses. The summand when $j = p$, counts the number of ways m memory accesses can be made to $(p+q)$ memory blocks of table $\mathcal{T}_i$, ensuring that there is at least 1 and at most p cache misses. Since we require exactly p cache misses (and not at most), we use the inclusion–exclusion principle to exclude the other events. □

Let $s_i = (p_{i,1}, p_{i,2}, \cdots, p_{i,\gamma}) \in \mathbb{S}_i$ be a particular sequence of cache misses. Then,

$$\mathsf{Pr}[\mathsf{S}_i = s_i] = \prod_{r=1}^{\gamma} \mathsf{Pr}[\mathsf{P}^m_{i,r} = p_{i,r} \mid \mathsf{Q}_{i,r} = q_{i,r}] , \tag{6.1}$$

where S_i is a random variable denoting the event of obtaining s_i from $\mathbb{S}_i$. This equation gives us the probability of obtaining a particular sequence of cache misses during the encryption from a single look-up table.

If multiple tables are used in the cipher, then the Cartesian product $(\mathbb{S}_1 \times \mathbb{S}_2 \times \cdots \times \mathbb{S}_g)$ is used to represent the cache misses in all tables. Take for example the $((4, 3, 1, 0, 0, \ldots) \times (5, 2, 0, 1, 1, \ldots))$. This means that in the first round, table one had four cache misses and table two five. In the second round, table one had three cache misses and table two five, and so on. The number of tables in the implementation is 2 (or $g = 2$). Since accesses to each table in the cipher is independent of the others we can write

$$\mathsf{Pr}[(s_1, s_2, \ldots, s_g)] = \prod_{i=1}^{g} \mathsf{Pr}[\mathsf{S}_i = s_i] \,, \tag{6.2}$$

where $(s_1, s_2, \ldots, s_g) \in (\mathbb{S}_1 \times \mathbb{S}_2 \times \cdots \times \mathbb{S}_g)$.

The *expected number of cache misses* that occur during an encryption when a single table is used (*i.e.,* $g = 1$), is determined from Eq. 6.1. This is given by,

$$\mu_{avg} = \sum_{s_1 \in \mathbb{S}_1} \mathsf{Pr}[\mathsf{S}_1 = s_1] \Big(\sum_{r=1}^{\gamma} p_{1,r} \Big) \qquad \text{when} \quad g = 1 \,. \tag{6.3}$$

When multiple tables are present (*i.e.,* $g > 1$), Eq. 6.2 can be used to find the *expected number of cache misses* as shown below

$$\mu_{avg} = \sum_{s \in \mathbb{S}_1 \times \mathbb{S}_2 \times \cdots \mathbb{S}_g} \mathsf{Pr}[s] \Big[\sum_{r=1}^{\gamma} \Big(\sum_{i=1}^{g} p_{i,r} \Big) \Big] \text{ when } \quad g > 1. \tag{6.4}$$

Equation 6.4 considers the cache misses in each table. In the next section we use Eqs. 6.1 and 6.2 to approximate the difference in execution times between two encryptions.

6.3 Average Execution Time of a Block Cipher

For memory intensive programs, the time to service cache misses largely dominates execution time. Block ciphers implemented with look-up tables fall into this category of programs. Due to this reason execution time of a block cipher can be estimated from the number of misses that have occurred in the cache.

Memory loads when there is a cache miss takes considerably longer than those that have a cache hit. Due to this, modern computer systems have several microarchitectural acceleration techniques incorporated in the cache—such as parallelization pipelining, nonblocking, and out-of-order servicing of cache misses. Parallelization and pipelining allow multiple cache misses to be serviced simultaneously, while out-of-order loading allows memory accesses to be performed in a sequence not strictly specified by the program. Nonblocking memory accesses allow other memory accesses to be done while a cache miss is being serviced (these architectural features are discussed in Sect. 3.2.2).

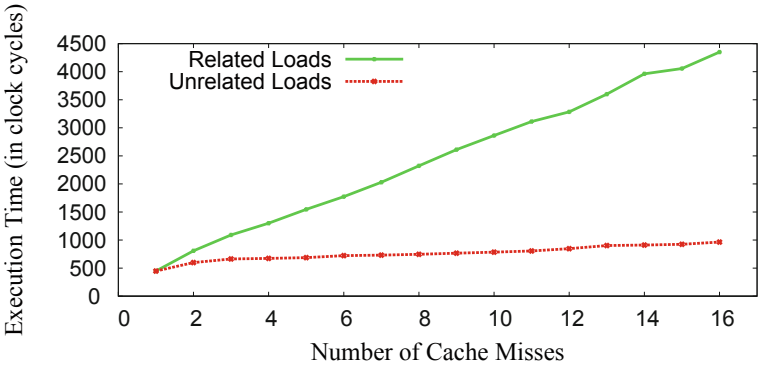


Fig. 6.2 Execution overheads for servicing cache misses on an Intel Dual Core processor (P6100). Related loads cannot be parallelized; therefore, the cache misses cannot be accelerated. Unrelated loads from memory are parallelizable; therefore, the time to service cache misses is less

To show how these microarchitectural techniques affect execution time we consider two programs. The programs make a series of memory loads, each load to a different memory location. In the first program, the address of a location loaded depends on the data read in the previous load (relative loads). In the second program, the memory loads are from independent locations (independent loads). We flush the cache each time to ensure that the required data is not present in the cache. Thus, the execution time for these programs depend mostly on the time required to service cache misses. Figure 6.2 shows the results on an Intel Dual Core (P6100) platform. The x-axis has the number of cache misses. This is equal to the number of loads, since the flushed cache ensures that every load to the table is a cache miss. The y-axis has the execution time for the program. In the case where the loads are related, memory loads cannot be parallelized or pipelined, or performed out of order. These accesses take longer compared to unrelated loads which can be accelerated by these microarchitectural features in the cache.

These two programs can be related to the memory loads made by block ciphers. In a typical block cipher implementation (such as the model in Fig. 6.1), each round has several memory accesses. The accesses within a single round are unrelated to each other. On the other hand, the diffusion layer in the cipher causes the data accessed in a round to be dependent on the previous round accesses. Due to this pattern of memory accesses, the following observations can be made.

1. Cache misses within a single round of the cipher can be accelerated due to the parallelism, pipelining, and out-of-order features in the cache.
2. Cache misses across rounds are done sequentially due to data dependencies.
3. The nonblocking feature in the cache allows hits within a single round to be done while the cache-misses are being serviced thus hiding the hit time.

For the block cipher, the time required to execute cache misses in one round comprising p_r cache misses can be estimated as $(\alpha + \beta(p_r))$. The constant α is the initial

latency for the first cache miss and β is the time required to service a cache miss. These constants are CPU-specific and also vary from one core in the system to another. In Fig. 6.2, the initial latency (α) was found to be approximately 180 and β, the slope of the line, was 16.

The total time required for servicing cache misses in the cipher is the sum of the time required for servicing cache misses in each round. The expected time to service cache misses is given as follows.

$$\widehat{\mathrm{t}}_{avg} = \sum_{s \in \mathbb{S}_1 \times \mathbb{S}_2 \times \cdots \times \mathbb{S}_g} \mathsf{Pr}[s] \Big(\sum_{r=1}^{\gamma} \widehat{\mathrm{t}}_{r,g} \Big) , \tag{6.5}$$

where $\widehat{\mathrm{t}}_{r,g}$ is the expected time taken for servicing cache misses from g tables in round r $(1 \leq r \leq \gamma)$[1]. This is given by

$$\widehat{\mathrm{t}}_{r,g} = \begin{cases} \Big(\alpha + \beta \big(\sum_{i=1}^{g} p_{i,r} \big) \Big) & \text{if } \sum_{i=1}^{g} p_{i,r} \neq 0 \\ 0 & \text{otherwise} \end{cases}$$

$p_{i,r}$ is the number of cache misses in round r due to table $\mathcal{T}_i$. Note that $\widehat{\mathrm{t}}_{r,g}$ is zero when all memory accesses in a round are cache hits. If in a round, even a single memory access results in a cache miss, then there is the large initial latency of α added to $\widehat{\mathrm{t}}_{r,g}$ for that round. Every subsequent cache miss causes the time to increase by a factor of β.

6.3.1 Estimating the Difference of Means

The time-driven cache adversary uses Algorithm 4.1 to identify collisions in specific memory accesses. When these collisions are detected, the following equation is satisfied.

$$\langle k_a \oplus d_a \rangle = \langle k_b \oplus d_b \rangle . \tag{6.6}$$

The collision results in an average execution time which is visibly different from the average case. If $\widehat{\mathrm{t}}_{collision}$ is the execution time when a collision is present, then the difference between the two expectations acts as a distinguisher. This is called the DOM or *difference of means* and is given by the following equation.

$$\widehat{\mathrm{d}} = \widehat{\mathrm{t}}_{collision} - \widehat{\mathrm{t}}_{avg} . \tag{6.7}$$

When Eq. 6.6 is satisfied, every encryption would have a collision during the memory load $T[k_b \oplus d_b]$. We refer to this as the *ever-present* collision.

[1] It may be noted that s has the form $((p_{1,1}, p_{1,2}, \ldots, p_{1,\gamma}), (p_{2,1}, p_{2,2}, \ldots, p_{2,\gamma}), \ldots \ldots, (p_{w,1}, p_{w,2}, \ldots, p_{w,\gamma}))$

For an *ever-present* collision in round r_c, the probability of obtaining cache misses (Theorem 6.1) is affected. Out of the m accesses that are made to the look-up table in that round, one always results in a cache hit due to the collision. Thus, the maximum number of cache misses in round r_c (i.e., upper bound for the random variable P^m_{i,r_c}, defined in Sect. 6.2) is $minimum(l, m-1)$ instead of $minimum(l, m)$, where l is the size of the table and m the number of accesses to the table in round r_c.

The following corollary shows the probability of cache misses in a round, given an ever-present collision.

Corollary 6.1 *The probability of having p_{i,r_c} cache misses in a round comprising of m memory accesses to table $\mathcal{T}_i$, given that one of the m accesses always results in a collision, is given by* $\Pr[\mathrm{P}^m_{i,r_c} = p_{i,r_c} | \mathrm{Q}_{i,r_c} = q_{i,r_c}, collision] = \Pr[\mathrm{P}^{m-1}_{i,r_c} = p_{i,r_c} | \mathrm{Q}_{i,r_c} = q_{i,r_c}]$ *when* $m > 1$, *while* $\Pr[\mathrm{P}^1_{i,r_c} = 1 | \mathrm{Q}_{i,r_c} = q_{i,r_c}, collision] = 0$ *when* $m = 1$.

Proof. There is a collision in round r_c which is ever present. Therefore, when $m = 1$, the probability of obtaining a cache miss is 0. Further, since we are only considering cache misses and not cache hits, the ever-present collision when $m > 1$, will not influence the probability. Therefore, obtaining p_{i,r_c} cache misses from m accesses given the ever-present collision is as probable as having p_{i,r_c} cache misses from $m-1$ random table accesses.

Since the cache miss probabilities in round r_c are affected by the ever-present collision, the probability of cache misses of subsequent rounds is also affected. Thus the effect of the ever-present collision spreads to the remaining rounds of the cipher as well. Further, the forced collision only affects the cache misses in the concerned table. For all other tables Theorem 6.1 still holds.

The estimation of $\widehat{\mathrm{t}}_{collision}$ can be done in exactly the same way as $\widehat{\mathrm{t}}_{avg}$. However, Corollary 6.1 has to be used for the round r_c. All other rounds would still follow Theorem 6.1.

6.4 DOM as a Security Metric

The time-driven adversary uses the difference between $\mathrm{t}_{collision}$ and t_{avg} to determine a collision. This difference is termed as the DOM (d). If $|d|$ is large, the adversary will find it easy to distinguish between $\mathrm{t}_{collision}$ and t_{avg} thus would require lesser measurements to identify the collision. As |d| decreases, it becomes increasingly difficult to distinguish between the two timings. The adversary may need more number of measurements, and sophisticated analysis techniques to make out the difference. To take an analogy, lets assume a scientist wants to distinguish between two colors. If the colors are far apart in the electromagnetic spectrum (for instance blue and red), then it would be easy to distinguish. He could probably do it with the naked eye. On the other hand, if the colors were almost similar (two shades of yellow for instance), then he would require more sophisticated tools, such as a spectrometer to make the difference.

We consider the absolute value of the DOM ($|\mathrm{d}|$) as a metric to compare the security of two cipher implementations. Larger the value of $|\mathrm{d}|$, the more easier the attack. Conversely, if $|\mathrm{d}| = 0$, then the collision cannot be identified, there by the attack will not be successful.

For a block cipher, the value of d is influenced by two factors.

F1 : The number of cache misses that occur during the execution of the cipher.
F2 : The influence of microarchitectural features in the cache such as pipelining, parallelization, and out-of-order servicing of cache misses. These components in the cache result in acceleration of memory accesses within a round.

The magnitude by which F1 and F2 affects the value of d depends on the implementation of the cipher, that is the parameters g, γ, m, and l (defined in Table 6.1). We demonstrate the effect of the cipher implementation by using Eq. 6.7. For the analysis we assume the values of (α, β) as (150, 16).

Figure 6.3a, b, and c plot the variation in the DOM, $\widehat{\mathrm{d}} = \widehat{\mathrm{t}}_{collision} - \widehat{\mathrm{t}}_{avg}$, with size of table ($l$) for different γ, g, and m respectively. When $l = 1$, there is no leakage because in every encryption the first memory access loads the entire table into the cache and all subsequent memory accesses would result in cache hits, thus $\widehat{\mathrm{d}} = 0$. For small values of $l > 1$, $\widehat{\mathrm{d}}$ is large and gradually reduces as l increases. For a particular value of l (other than 1), $\widehat{\mathrm{d}} = 0$. This is the ideal table size for the cipher as it implies that the distributions cannot be distinguished. We denote this value of l by l_{ideal}. Implementations of the cipher having $l < l_{ideal}$ are called *small-table implementations* while implementations having $l > l_{ideal}$ are called *large-table implementations*.

In small-table implementations, the size of the look-up table is small compared to the number of accesses made to it. With significant probability the entire table gets loaded into cache in every encryption. Thus every encryption will have the maximum permissible cache misses (which is the number of memory blocks in the table l). Even with a collision satisfied, l cache misses still occur. Thus F1 has no contribution in the leakage and it is only F2 which causes the leakage. The ever-present cache hit in round r_c ($1 \leq r_c \leq \gamma$) restricts acceleration and results in $\widehat{\mathrm{t}}_{collision} > \widehat{\mathrm{t}}_{avg}$. As a result $\widehat{\mathrm{d}}$ is positive. This means that a cache hit due to the collision actually slowed down the encryption. This is counter intuitive. Figure 6.4 shows an example of how this could happen.

In large-table implementations, the probability that the entire table gets loaded into cache is almost zero. When a collision is ever present, due to Eq. 6.6 being satisfied, there is a reduction in the number of cache misses, which in turn results in faster encryptions. Subsequently $\widehat{\mathrm{t}}_{collision}$ is less than $\widehat{\mathrm{t}}_{avg}$ and $\widehat{\mathrm{d}}$ is negative. In this case both F1 and F2 contribute to the information leakage. F2 affects $\widehat{\mathrm{d}}$ in the positive direction while F1 affects $\widehat{\mathrm{d}}$ in the negative direction. However, the impact of the reduced miss is higher compared to the acceleration obtained, thus F1 dominates F2.

When $l = l_{ideal}$, the effects of F1 and F2 cancel each other resulting in $\widehat{\mathrm{d}} = 0$, therefore protected against this form of nonprofiled time-driven cache attacks. This phenomenon was practically visualized on several modern microprocessors including Intel Atom, Dual Core, Core 2 Duo, i3, and Pentium 4. However, on the older Pentium 3 processors, the microarchitectural effects due to F2 were not observed.

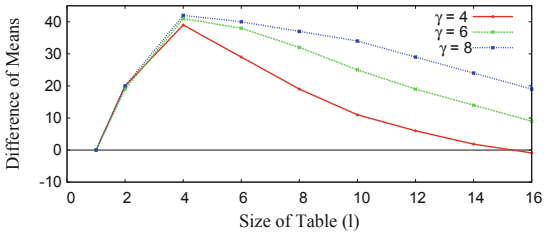
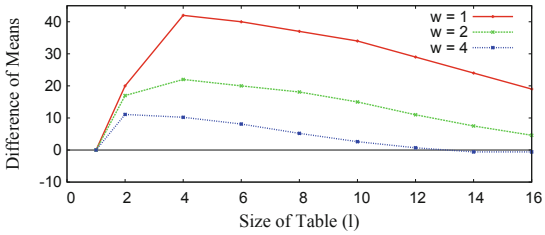
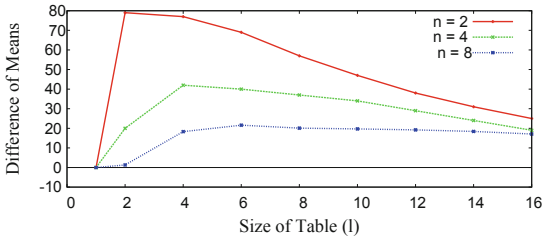


Fig. 6.3 The effect of different cipher models on the difference of means (d). All graphs are plotted with the size of the look-up tables on the x-axis. The value of d reduces as the look-up table size increases

a Cipher models with different numbers of rounds and $m = 4$, $g = 1$. The value of d increases as the number of rounds in the cipher increase.



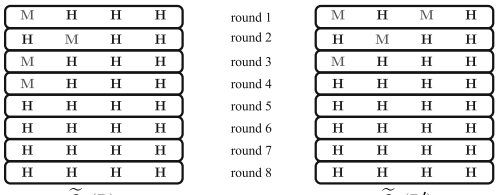
b Cipher models with different number of tables and $m = 4$, $\gamma = 8$. The value of d reduces as the number of tables used in a cipher implementation increases.



c Cipher models with different number of table accesses per round and $g = 1$, $\gamma = 8$. The value of d reduces as the number of table accesses per round increase.

6.5 Application of the Model

The model for predicting the DOM can be used to compare the security of cipher implementations and also select the best way to implement a cipher. We compare four standard block cipher implementations of AES, CLEFIA, data encryption standard (DES), and CAMELLIA. In the second part of the section we evaluate various implementation options for CLEFIA and CAMELLIA with the aim of finding implementations more resistant against time-driven cache attacks.



Consider a cipher implementation $\widetilde{\mathcal{E}}_{\mathrm{K}}$ with a cipher model $w = 1$, $\gamma = 8$, $l = 4$, and $n = 4$. Consider encryptions of two different plaintexts P and P' : $\widetilde{\mathcal{E}}_{\mathrm{K}}(\mathrm{P})$ and $\widetilde{\mathcal{E}}_{\mathrm{K}}(\mathrm{P}')$. For each encryption, the figure shows the result of memory accesses to the look-up table. A memory access that resulted in a cache hit is represented by **H** while a memory access that resulted in a cache miss is represented by **M**. It may be noted that both encryptions result in equal number of cache misses (i.e. 4). However the distribution of the misses differ. In $\widetilde{\mathcal{E}}_{\mathrm{K}}(\mathrm{P})$ there is at most a single cache miss in each round, while in $\widetilde{\mathcal{E}}_{\mathrm{K}}(\mathrm{P}')$ there exists a round with 2 cache misses. Due to the MAF in the cache, the pair of cache misses in round 1 during the encryption of $\widetilde{\mathcal{E}}_{\mathrm{K}}(\mathrm{P}')$ gets serviced faster compared to any pair of the cache misses that occur during the execution of $\widetilde{\mathcal{E}}_{\mathrm{K}}(\mathrm{P})$. This results in a faster execution for $\widetilde{\mathcal{E}}_{\mathrm{K}}(\mathrm{P}')$ compared to $\widetilde{\mathcal{E}}_{\mathrm{K}}(\mathrm{P})$ even though both encryptions have the same number of cache misses.

Fig. 6.4 Example of collisions in a cipher that uses a small look-up table

The verification of the results are done using the Intel Xeon E5606 platform. Refer to Table A.1 in the appendix for details on the system's cache memory configuration. However, the cache configuration alone is not sufficient to apply the model developed. Additional parameters of α and β are required as well. The values of α and β found are also depicted in the table for each platform.

6.5.1 *Comparing Cipher Implementations*

Figure 6.5 shows the DOM obtained for the four cipher implementations considered in Table 6.2. Each implementation was analyzed theoretically to obtain the modeled DOM for the platforms and then the empirical DOM was obtained after 2^{20} measurements.

The histogram shows that the modeled DOM matches the empirical results closely. The OpenSSL implementation of AES is the most secure against time-driven cache attacks compared to the other cipher implementations analyzed as it has the smallest absolute value of DOM. The security of the AES implementation is due to the large number and size of tables, which are implemented as T-tables (Refer Sect. 2.2.1.1). As seen in Fig. 6.3, large number and size of tables result in smaller DOM thus more security. Further, the tables in OpenSSL's AES implementation are T-tables, which reduces the diffusion to just 16 ex-ors per round intertwined with the table accesses.

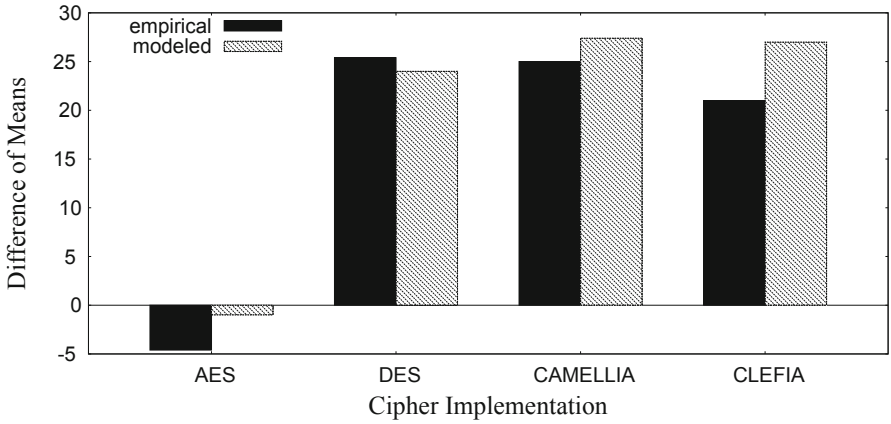


Fig. 6.5 Difference of means obtained from the model and empirically for the cipher implementations in Table 6.2 on Intel Xeon 5606

The T-tables allow cache misses from one round to overlap with another, there by resulting in smaller α (approximately 10). For $\alpha = 10$, the value of l_{ideal} is close to 16, which is the size of the tables in OpenSSL AES on the Intel Xeon processors. This *close-to-ideal* size of the tables further contributes to the small DOM.

6.5.2 Choosing the Right Implementation

In this part of the section we use the DOM model to judge the security of a set of implementation options for CLEFIA and CAMELLIA, with the hope of obtaining more secure implementations for these ciphers compared to their implementations in Table 6.2. Tables 6.3 and 6.4 show the the DOM obtained from the mathematical model for both platforms evaluated (E5606 and E5345). This DOM can be used to

Table 6.3 Implementation options for CLEFIA

No.	$\mathbf{l}^a$	γ	**m**	**w**	Table size (in bytes)	Leaked bits	DOM $\widehat{\mathrm{d}}$
CL1[b]	4	18	4	2	256	2	27
CL2	8	18	4	2	512	3	25
CL3[c]	16	18	1	8	1024	4	−1
CL4	4	18	2	4	256	2	27
CL5	8	18	2	4	512	3	23
CL6	16	18	2	4	1024	4	10

DOM difference of means
[a] In all cases we assume a cache with a cache line of 64 bytes
[b] This corresponds to the reference implementation (Table 6.2)
[c] This corresponds to CLEFIA implemented with *T*-tables

Table 6.4 Implementation options for CAMELLIA

No.	l[a]	γ	m	w	Table size (in bytes)	Leaked bits	DOM $\widehat{d}$
CA1[b]	4	18	2	4	256	2	36
CA2	8	18	2	4	512	3	31
CA3	16	18	2	4	1024	4	12
CA4	4	18	8	1	256	2	29
CA5	8	18	8	1	512	3	32
CA6	16	18	8	1	1024	4	31

DOM difference of means
[a] In all cases we assume a cache with a cache line of 64 bytes
[b] This corresponds to the reference implementation (Table 6.2)

relatively rank the implementations in the order of their security against nonprofiled time-driven cache attacks. Stronger implementations are those with a DOM closer to zero. For CLEFIA (Table 6.3), we can thus conclude that CL3 is the most secure against and CL6 is the next best. The remaining implementations are insecure. For CAMELLIA (Table 6.4), the implementation CA3 turns out to be most secure with respect to the other implementations investigated.

To verify the correctness of the predictions, we subject the identified implementations to time-driven cache attacks using Algorithm 4.1. In all cases the first pair of accesses to the same table were targeted. We gradually increase the number of

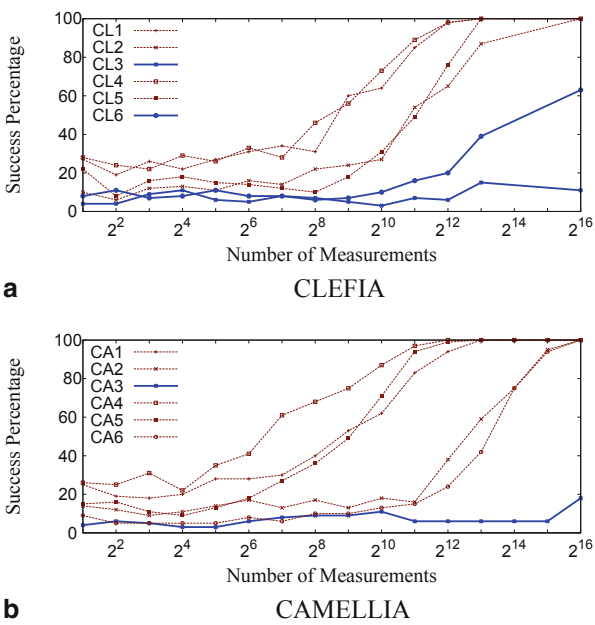


Fig. 6.6 Success of an attack for various implementations of CLEFIA (Table 6.3) and CAMELLIA (Table 6.4) on Intel Xeon 5606

timing measurements made and determine the success rate with which the bits of the ex-or of the key bytes are determined. Figure 6.6 shows this success percentage for CLEFIA and CAMELLIA on both platforms after performing 100 tests. As predicted, CL3 and CL6 are the most secure implementation for CLEFIA, since the attacker achieves minimum success in obtaining the key compared to the other implementations. Similarly, the prediction CA3 for CAMELLIA is the most secure as was predicted in Table 6.4.

6.6 Conclusion

The DOM is used as a distinguisher in time-driven cache attacks. The number of cache misses that occur during the execution affects the DOM negatively, while microarchitectural components in the cache such as out-of-order, parallel, and pipelined servicing of cache misses affect the DOM positively. Generally block ciphers implemented with small look-up tables have a positive DOM while those with large look-up tables have a negative DOM. It is also possible to implement the block cipher with a DOM of 0 or close to 0. Such implementations would be more resistant against time-driven cache attacks without the use of any explicit countermeasures. These implementations provide security with no performance overheads.

Chapter 7
Profiled Time-Driven Cache Attacks on Block Ciphers

In 2005, D. J. Bernstein developed a timing attack capable of retrieving the Advanced Encryption Standard (AES) secret key [4]. Unlike the previous attacks discussed so far, Bernstein's attack has two phases: a profiling phase followed by an attack phase. During the *profiling phase*, the attacker learns the characteristics of the system by building a timing profile called *template* using a key which is known to her. The template captures all the timing characteristics when AES is executed. With this template, any other secret key used in the AES implementation on that system can be attacked. During the *attack phase*, another timing profile is built for the secret key. A statistical comparison of this timing profile with the template reveals the secret key. This chapter provides details of the attack and analyzes the information leaked.

7.1 Bernstein's Cache Timing Attack

7.1.1 Building a Timing Profile

The AES input has 16 bytes as seen in Sect. 2.2.1. To build the timing profiles we first start by choosing random inputs (say $\mathbf{x} = (x_0 \parallel \cdots \parallel x_i \parallel \cdots \parallel x_{15})$, $0 \leq i \leq 15$), invoking AES and measuring the execution time. This is repeated several times with different inputs. We define 16 arrays denoted $\mathcal{A}_i$ $(0 \leq i \leq 15)$ each of size 256, which stores the average execution time. The element $\mathcal{A}_i[j]$ $(0 \leq j \leq 255)$ in an array stores the average execution time when the ith byte of the plaintext was j. We then find the overall average execution time (denoted t_{avg}) and compute deviations as follows: $\mathcal{D}_i[j] = \mathcal{A}_i[j] - t_{avg}$, where $0 \leq i \leq 15$ and $0 \leq j \leq 255$. The ith deviation array ($\mathcal{D}_i$) is called the *timing profile* for the ith input byte. We thus have 16 deviation arrays; one for each input byte. Algorithm 7.1 shows how the templates are built for AES. The array $\mathcal{T}$ accumulates the encryption time, while $\mathcal{C}$ counts the number of times a particular input is obtained. Figure 7.1 shows the timing profile for the first plaintext byte of AES. It plots the value of x_0 on the x-axis

C. Rebeiro et al., *Timing Channels in Cryptography,*
DOI 10.1007/978-3-319-12370-7_7

Algorithm 7.1: Timing Profile

Output: $\mathcal{D}$
begin
  $\mathcal{C}_i[j] \leftarrow 0$ for $0 \le i \le 15$ and $0 \le j \le 255$
  $\mathcal{T}_i[j] \leftarrow 0$ for $0 \le i \le 15$ and $0 \le j \le 255$
  for *at-least* 2^{16} *times* **do**
    $\mathbf{x} \leftarrow$ Random plaintext
    $t_1 \leftarrow$ start time
    $AES_{\mathbf{k}}(\mathbf{x})$
    $t_2 \leftarrow$ end time
    $\mathcal{C}_i[x_i] \leftarrow \mathcal{C}_i[x_i] + 1$ for $0 \le i \le 15$
    $\mathcal{T}_i[x_i] \leftarrow \mathcal{T}_i[x_i] + (t_2 - t_1)$ for $0 \le i \le 15$
  end
  $\mathcal{A}_i[j] \leftarrow \mathcal{T}_i[j]/\mathcal{C}_i[j]$ for $0 \le i \le 15$ and $0 \le j \le 255$
  $t_{avg} \leftarrow \sum_{i,j} T_i[j] / \sum_{i,j} C_i[j]$
  $\mathcal{D}_i[j] \leftarrow \mathcal{A}_i[j] - t_{avg}$
  return $\mathcal{D}$
end

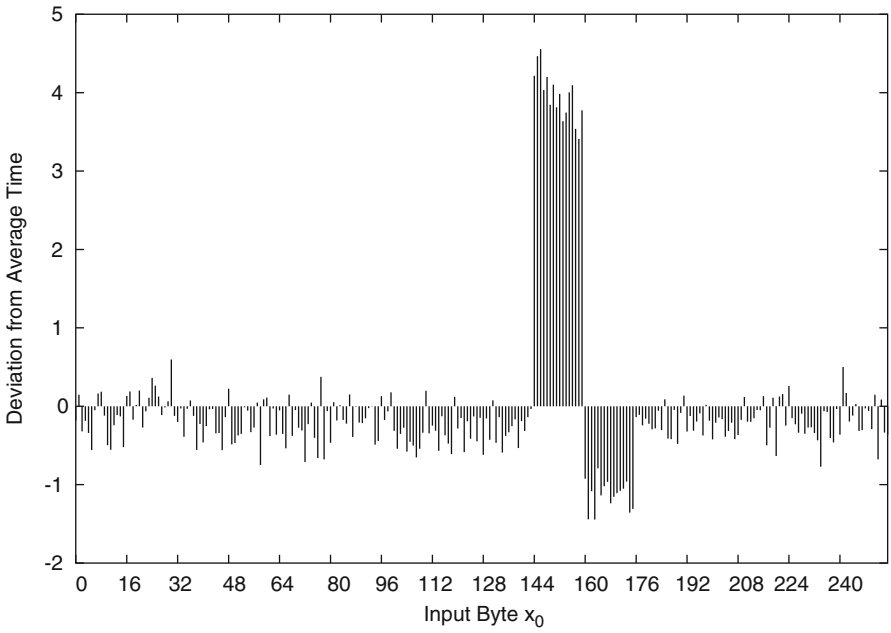


Fig. 7.1 Timing profile for $\mathcal{D}_0$ vs. x_0 for AES on an Intel Core 2 Duo (E7500)

and $\mathcal{D}_0[x_0]$ on the y-axis. To attack a key byte, two such timing profiles are required—one for a known key and the other for the unknown key.

7.1.2 Extracting Keys from Timing Profiles

There are 16 timing profiles created ($\mathcal{D}_0, \mathcal{D}_1, \ldots, \mathcal{D}_{15}$)—one for each input byte. Each vertical line in a timing profile provides an average execution time deviation $\mathcal{D}_i[j]$, when the ith input byte is fixed at j and all other bytes vary randomly. As seen in the AES specification (Algorithm 2.1), the ith input corresponds to the ith look-up table access in the first round. This access is at an index $s_i^{(0)} = x_i \oplus k_i^{(0)}$ in the look-up table. Thus the time deviation measures the deviation when $s_i^{(0)}$ is fixed and all other intermediate values in the cipher vary randomly. With respect to the look-up tables in the implementation, the table access made by $s_i^{(0)}$ is the only fixed operation in the entire encryption. All other table look-up operations are at random locations. When the average of several million encryptions are considered, the influence of all the varying table accesses in the execution time cancel out leaving behind only the effect of the constant table access. Thus, $\mathcal{D}_i[s_i^{(0)}]$ is a characteristic deviation of execution time of $s_i^{(0)}$.

Now consider two timing profiles for the ith byte ($0 \leq i \leq 15$): one from a known key (denoted $\hat{k}_i^{(0)}$) and another from an unknown key (denoted $\tilde{k}_i^{(0)}$). The deviation $\mathcal{D}_i[s_i^{(0)}]$ is an invariant and a characteristic of the element in the table accessed (i.e., at index $s_i^{(0)}$). There are two ways to access this element in the table using the i-th input byte. Using the known key $\hat{k}_i^{(0)}$ the access can be obtained from $\hat{x}_i \oplus \hat{k}_i^{(0)}$ for some $\hat{x}_i$ ($0 \leq \hat{x}_i \leq 255$). Second, using $\tilde{k}_i^{(0)}$ the access can be obtained from $\tilde{x}_i \oplus \tilde{k}_i^{(0)}$ for some $\tilde{x}_i$ ($0 \leq \tilde{x}_i \leq 255$). Thus with reference to the deviation of time, the following relation would be met

$$\mathcal{D}_i[\tilde{x}_i \oplus \tilde{k}_i^{(0)}] = \mathcal{D}_i[\hat{x}_i \oplus \hat{k}_i^{(0)}] \,. \tag{7.1}$$

With respect to the timing profiles, the invariant has shifted from $\hat{x}_i$ in the timing profile of the known key to $\tilde{x}_i$ in the timing profile of the unknown key. In the attack we determine this shift and then compute the unknown key as follows: $\tilde{k}_i^{(0)} = \tilde{x}_i \oplus \hat{x}_i \oplus \tilde{k}_i^{(0)}$.

To determine the shift, the unknown key byte is guessed (say $kguess$) and for every possible value of x ($0 \leq x \leq 255$), a correlation coefficient is computed as follows:

$$CC_{kguess} = \sum_{x=0}^{255} \mathcal{D}_i[x \oplus \hat{k}_i^{(0)}] \times \mathcal{D}_i[x \oplus kguess] \tag{7.2}$$

This equation essentially computes the correlation between the timing profile of the known key and a shifted version of the timing profile of the unknown key. The shift being $kguess$. There are thus 256 values of CC_{kguess} (one for each $kguess$). The key guess corresponding to the maximum value of CC_{kguess} is most likely the unknown secret byte of the key ($\tilde{k}_i^{(0)}$).

7.2 Causes of Information Leakage

The vector $\mathcal{D}_i$, which represents a timing profile, has 256 different entries. The only difference between each entry in the vector is that x_i $(0 \leq i \leq 15)$ changes. Each x_i represents a unique element in a look-up table that is accessed in the first round during the cipher's execution. The element is accessed at the index $x_i \oplus k_i^{(0)}$. Since the look-up table has 256 entries, the vector $\mathcal{D}_i$ represents execution time deviation corresponding to each element in the table.

Timing variations in the profiles can be attributed due to two sources: intrablock and interblock. *Interblock* sources cause timing variations between memory blocks. In other words if two values of x_i result in table accesses in different memory blocks, the execution time is likely to be different. However, if the two values of x_i fall in the same memory block then they would have the same execution time. Thus interblock sources can only distinguish between accesses in the look-up table that fall in different memory blocks.

Intrablock sources result in execution time of the cipher that is different for each element in the table. Thus each value of x_i is likely to have a different execution time even if the corresponding access at the index $x_i \oplus k_i^{(0)}$ are to adjacent elements in the look-up table. Thus, intrablock sources can distinguish between every access made to the look-up table.

During the final postprocessing phase, shifts between the template and the attack profile are determined. The shift between the two profiles are used to determine $k_i^{(0)}$. The shifts could take any value from 0 to 255. Consequently $k_i^{(0)}$ can take values in this range. However, shifts can only be identified if there are variations in the timing profile. With inter-block sources, there is no timing variation within a memory block. As a result, shifts in a memory block cannot be identified. As a result, attacker would be able to identify only the most significant bits of $k_i^{(0)}$. With intrablock sources, each value of x_i is likely to have different execution time. Therefore, this source can be used to identify all bits of $k_i^{(0)}$.

In the remainder of this section we discuss the various sources that affect the average execution time. We discuss how an attack on OpenSSL's implementation of AES[1], is primarily due to *interblock* sources, while an attack on an implementation of CLEFIA[2], is due to *intrablock* sources. As a result, the bits of the key recovered from the AES attack are restricted to the higher bits. On the other hand, all bits of CLEFIA's key are recovered. This results in better success rates for the CLEFIA attack.

Interblock Sources: A primary interblock source is conflict misses. These cache misses occur due to memory accesses which evict recently accessed data, thus resulting in misses. Conflict misses are ideally sporadic and occur at random cache

[1] OpenSSL version 0.9.8a, http://www.openssl.org details present in Sect. 2.2.1.

[2] CLEFIA Reference code, http://www.sony.net/Products/cryptography/clefia/download/data/clefa_ref.c, details present in Sect. 2.2.1.

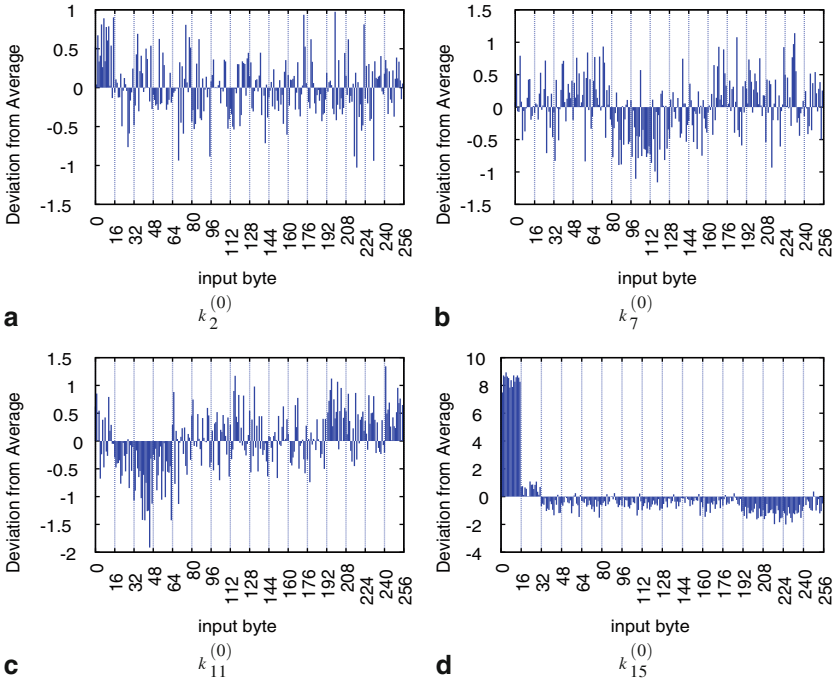


Fig. 7.2 AES timing profiles for four key bytes after 2^{27} encryptions on an Intel Core 2 Duo (E7500) system

sets. Such conflict misses would cause random variations in the timing profiles, thus not posing a threat. But as pointed out by Neve, Seifert, and Wang [1; 2], conflict misses can also be periodic. Periodic conflict misses cause cache misses in the same cache set at regular intervals of time. This periodic form of conflict misses affect the average execution time, because they evict the same bytes from the cache in every encryption. Additionally, a conflict miss requires servicing of a cache miss, which takes considerably long time. Thus conflict misses have a large impact on the information leakage.

Large look-up tables are more likely to be affected by conflict misses, since they occupy a larger region of the cache memory. Thus the AES implementation, which uses tables four times larger than the CLEFIA implementation, is more likely to be affected by conflicts. This can be seen in the timing profiles for an AES attack (some of the profiles are shown in Fig. 7.2). Each table in the AES implementation has 256 values spread over 16 contiguous memory blocks (depicted by the vertical grid in Fig. 7.2), with each block holding 16 elements. Each point in the timing profile has the characteristic time for the access to a unique element in the table. A conflict miss will equally affect the characteristic time for all elements in a block. As a result of this, the correlation in the final phase of the attack produces the same result for shifts that are smaller than a memory block. Since each shift represents a key candidate,

Table 7.1 Ten most likely keys obtained from the AES attack after 2^{27} encryptions on an Intel Core 2 Duo (E7500) system

key	Correct key	Ten most likely keys for each byte									
$k_0^{(0)}$	**11**	4e	47	41	4a	46	4c	48	45	4f	44
$k_1^{(0)}$	**22**	05	**22**	c2	*2f*	ca	33	e1	06	*23*	c9
$k_2^{(0)}$	**33**	**33**	*38*	*3b*	*3a*	*34*	*37*	*39*	0c	*3f*	a7
$k_3^{(0)}$	**44**	83	89	8a	81	*41*	8b	84	*46*	*4b*	*4a*
$k_4^{(0)}$	**55**	d1	de	d9	a8	d0	d3	aa	a5	a0	a1
$k_5^{(0)}$	**66**	8f	52	c3	7a	2b	50	1a	23	f6	4a
$k_6^{(0)}$	**77**	79	*73*	*78*	*74*	**77**	*7e*	*7f*	*75*	8d	8e
$k_7^{(0)}$	**88**	*8e*	*87*	*8f*	*80*	*8a*	*86*	*89*	*8d*	*8b*	**88**
$k_8^{(0)}$	**99**	**99**	39	83	*90*	ba	1e	7a	af	70	13
$k_9^{(0)}$	**aa**	b4	e2	7b	e8	b1	c8	53	7a	79	bb
$k_{10}^{(0)}$	**bb**	65	57	5f	*b2*	24	*b6*	60	25	5e	80
$k_{11}^{(0)}$	**cc**	*c6*	*c2*	*ce*	*ca*	*cb*	**cc**	*c1*	*c0*	14	*cf*
$k_{12}^{(0)}$	**dd**	53	5b	50	52	49	58	5d	51	*d1*	48
$k_{13}^{(0)}$	**ee**	7c	*e0*	4e	98	94	*eb*	*e5*	d7	b3	3b
$k_{14}^{(0)}$	**ff**	ea	*fd*	*fb*	3a	e1	a4	e9	03	*f1*	**ff**
$k_{15}^{(0)}$	**00**	*05*	*01*	*06*	*02*	*04*	*08*	*03*	*0a*	*00*	*0c*

the key space is partitioned based on the size of the memory block, and all keys in the same partition are ranked together. Table 7.1 shows the top ten candidate keys for each byte for an AES attack. The third column is ordered from the most likely key to the least likely of the ten keys. The correct key is represented in bold red, while the keys which fall in the same cache line are emphasized in blue. As an example consider the key $k_6^{(0)}$, whose correct value is 77. The keys which map to the same memory block are 70 to $7f$. These have ranks which are close together. In most cases the attack on AES fails to distinguish between these keys. This results in a low success rate.

Another intersource of leakage is the hardware prefetcher. Most superscalar systems have a hardware prefetcher which anticipates future memory accesses and prefetches them into the cache in order to reduce the miss rate. The prefetcher works by detecting patterns in memory accesses. In cipher implementations, the interesting memory accesses (i.e., to the look-up tables) are expected to be at random locations, thus following no deterministic pattern. However, prefetchers are still capable of prefetching parts of the look-up table albeit inefficiently. This can nevertheless result in information leakage. A detailed analysis of this leakage source is done in Sect. 7.4.

Intrablock Sources: Microarchitectural components in the processor and memory such as the cache banks may cause a small variation in execution time of the cipher. Bernstein gives an example of how recently occurring stores to certain memory locations affect load time. Another example is loads from memory, which cause conflicts in cache banks. The effects of these microarchitectural components in the timing profile is more profound when the size of the look-up tables used in the implementation is small. In such cases conflict misses are less likely and every encryption is likely

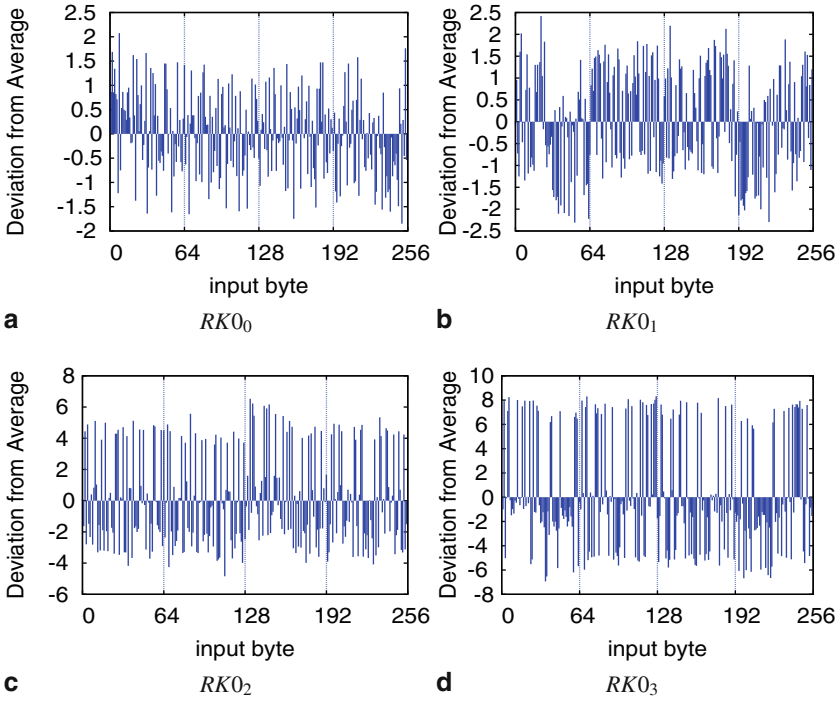


Fig. 7.3 CLEFIA timing profiles for four key bytes after 2^{27} encryptions on an Intel Core 2 Duo (E7500) system

to have the same number of cache misses. Minor variations in execution time due to the microarchitectural components become important. Some of these variations are at the word level and not the block level. These can distinguish keys within a block. Sources which cause such variations are called intrablock sources.

CLEFIA uses two small tables of 256 bytes (each occupying 4 memory blocks and each block holding 64 elements). Every encryption is likely to load the entire table into the cache. Also the effect of conflict misses is less due to the smaller tables. Thus leakage due to microarchitectural features (including the prefetcher) becomes important. This can be seen from the timing profiles for CLEFIA (Fig. 7.3), where (unlike the AES profiles in Fig. 7.2), no distinct patterns are present that can identify a memory block (each memory block is represented by the vertical grid lines). As a consequence, the keys obtained from the correlation phase are not necessarily ordered based on the memory blocks. This can be seen in Table 7.2 which shows the ten most likely key candidates for an attack on CLEFIA ordered from the most likely to the least likely of the ten. The correct key is represented in bold red and is always ranked first for this particular trial, while keys which fall in the same block are emphasized in blue. The smaller impact of interblock leakage sources is evident from the results.

Table 7.2 Ten most likely keys obtained from the CLEFIA attack after 2^{27} encryptions on an Intel Core 2 Duo (E7500) system

key	Correct key	Ten most likely keys for each byte									
$RK0_0$	**f4**	**f4**	*e2*	*c1*	*eb*	52	18	*e1*	*d7*	14	44
$RK0_1$	**d0**	**d0**	52	*f0*	*df*	46	51	*d8*	44	*f2*	*d7*
$RK0_3$	**6a**	**6a**	*5f*	94	92	e8	*48*	*6c*	*75*	a9	*b6*
$RK1_0$	**ca**	**ca**	a7	5b	40	54	52	bf	58	51	53
$RK1_1$	**7b**	**7b**	*46*	db	d1	c6	c4	*52*	*56*	8f	*79*
$RK1_2$	**91**	**91**	13	5a	*8c*	f2	14	64	*a8*	f6	36
$RK1_3$	**60**	**60**	ab	07	*68*	c5	ec	9c	*78*	e9	16
$RK2_0 \oplus WK0_0$	**fe**	**fe**	*f8*	00	06	*ec*	14	11	1c	*f6*	1b
$RK2_1 \oplus WK0_1$	**57**	**57**	*51*	*62*	a7	*5a*	f7	*64*	24	e1	9f
$RK2_2 \oplus WK0_2$	**3c**	**3c**	ea	c5	eb	*3d*	8c	be	92	*11*	ec
$RK2_3 \oplus WK0_3$	**80**	**80**	51	02	58	57	3c	d8	*89*	10	74
$RK3_0 \oplus WK1_0$	**6b**	**6b**	*7b*	*42*	90	*6f*	a3	*56*	d6	3d	a9
$RK3_1 \oplus WK1_1$	**40**	**40**	*4a*	b1	88	fd	92	16	2b	05	13
$RK3_2 \oplus WK1_2$	**16**	**16**	*05*	94	fd	45	6b	b9	*15*	f8	6e
$RK3_3 \oplus WK1_3$	**36**	**36**	f2	42	a8	ad	86	80	c5	*1b*	*34*
$RK4_0$	**7e**	**7e**	e0	fe	e8	01	11	ff	07	1c	12
$RK4_1$	**32**	**32**	*2f*	*34*	*26*	*38*	*31*	*35*	*3f*	*3e*	*29*
$RK4_2$	**50**	**50**	*5d*	00	a0	81	f0	*65*	82	b0	03
$RK4_3$	**e1**	**e1**	0e	37	*dc*	63	*cc*	*c8*	*e5*	89	77
$RK5_0$	**eb**	**eb**	9b	*da*	85	1e	*f8*	3e	*fe*	4c	99
$RK5_1$	**11**	**11**	*24*	e9	ef	*33*	93	cd	*0e*	d2	*17*
$RK5_2$	**47**	**47**	37	92	f8	99	8c	bb	34	b2	*52*
$RK5_3$	**35**	**35**	b7	*38*	7f	e7	5f	*31*	e8	8b	ed

Execution Time of Initial Rounds: The timing profile logs the variation in encryption time as a result of an access to a look-up table in the cipher's implementation. Besides this *specific* memory operation, all other operations done during the execution of the cipher can be considered as noise. The influence this operation has on the execution time of the cipher can determine how efficiently a key is recovered. A large influence implies that the execution time for the *specific* memory operation significantly affects the total encryption time, which can be therefore easily distinguished. A small influence on the other hand implies that execution time for the *specific* memory operation cannot be easily distinguished from the total encryption time.

The *specific* memory operation is in the first round for the AES attack and in the first three rounds for the CLEFIA attack. Thus the influence of the memory operations in these rounds on the encryption time will determine how efficiently the key can be recovered. Figure 7.4 shows the percentage influence in the encryption time for the first nine rounds of CLEFIA and AES.

Due to the small tables used in the CLEFIA implementation, the first three rounds of the cipher load the entire table into the cache. Subsequently, cache misses are most likely in the first three rounds. These rounds will have both hits and misses, while all remaining rounds will generally have only cache hits. As a result, over 95 % of the variations in the encryption time are due to the first three rounds. Since these are the rounds targeted during the attack, the variations due to memory operations from these rounds significantly influence the encryption time and are easily captured.

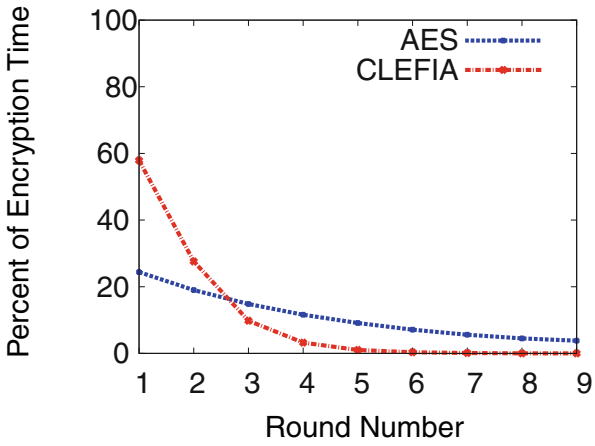


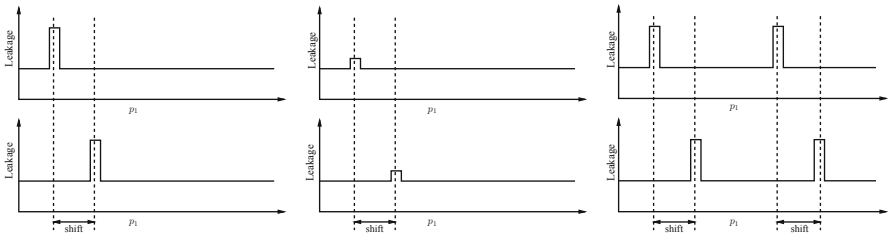
Fig. 7.4 Percentage of encryption time by each round of AES and CLEFIA (only first nine rounds shown)

AES on the other hand has large tables therefore cache misses are spread over a larger number of rounds. As a result, the targeted round (the first round), makes up only 25 % of the total encryption time. Thus time variations due to memory accesses made in this round do not influence significantly the encryption time, thus difficult to capture. This makes recovery of an AES key more difficult than that of a CLEFIA key and more number of AES encryptions need to be monitored in order to recover more bits of the key. In the next section we provide a generalized profiled time-driven cache attack, in order to boost the success rate of the attacks without increasing the number of encryptions required.

7.3 Quantifying Information Leakage in a Timing Profile

Bernstein's attack uses correlation to determine the shift between the timing profile for the known key and that of the unknown key. Each timing profile has 256 points corresponding to $x_i \oplus k_i$, where x_i varies from 0 to 255. Variations in the values at these points are characteristic features of the timing profile. These features are matched during the correlation. The efficiency of the attack is determined by how well features of one profile match with the other for the correct shift. An efficient attack would give a correlation close to 1 when the right shift is applied, while a correlation close to 0 when the shift is wrong.

The attack would fail if the correlation values obtained are the same irrespective of the shifts in the timing profile. For instance, consider a constant time implementation of a cipher. The average deviations in execution time would be zero irrespective of the inputs. The timing profiles for the known key and unknown key would be just a set of zeros. The correlation computed between these profiles is the same irrespective of the shift applied. A constant time implementation would therefore be able to resist the attack since the keys are not distinguishable. Thus the unique features in the



a Single, Large Deviating Distribution **b** Single, Small Deviating Distribution **c** Multiple Deviating Distributions

Fig. 7.5 Cache profiles with single or multiple deviating distributions

timing profile influences the efficiency of the attack. More the unique features in the profile, more efficient the attack. For instance, the timing profile in Fig. 7.5a is likely to result in more efficient attacks than that of Fig. 7.5b, and less efficient attacks compared to Figure 7.5c. As an other example, consider the attacks on AES and CLEFIA. The AES timing profile, which is more influenced by interblock leakage sources, has less unique features compared to the CLEFIA timing profile. As a result, AES attacks are less efficient compared to CLEFIA attacks.

In order to quantify the unique features in a timing profile, the metric in Eq. 7.3 can be used.

$$\mathcal{L} = \sum_{\substack{\forall pairs\ of \\ j\ and\ j'}} |\mathcal{D}[j] - \mathcal{D}[j']| \tag{7.3}$$

The index of the summation, j, ranges from 0 to 255. There are two parts in Eq. 7.3: (a) the difference term and (b) the summation over all pairs of distributions in the cache profile. The difference term quantifies how much one point in the profile differs from another. When two points on the timing profile are the same, they have a difference zero and cannot be distinguished. If the two points have different values, the difference is positive, and they can be distinguished. The distinguishability becomes easier as the difference between the points increase. The summation in Eq. 7.3 quantifies cases where the profile has more variations in the distributions leading to more information leakage. In the next section we show how the information leaked through hardware prefetching can be analyzed.

7.4 Information Leakage due to Hardware Prefetching

One way to prevent Bernstein's attack is to have the cipher execute at constant time. There are various strategies that have been adopted to provide constant time implementations. For example, the use of specialized instructions, such as Intel's AES-NI [3], eliminate memory-based look-up tables resulting in constant time execution of the cipher. However, these instructions are specific for the AES block

Algorithm 7.2: *SP* : Sequential Prefetching Algorithm

```
   Input: The access to memory block at address t_i
1  begin
2      if (t_i not in cache) or (t_i was prefetched and this is the first access to t_i) then
3          if t_{i+1} not present in the cache then
4              prefetch t_{i+1}
5          end
6      end
7  end
```

cipher, leaving a large number of other block ciphers still vulnerable. An alternate direction is to build platforms which inherently protect against cache attacks. This approach is more general and would be usable with a variety of ciphers. A step towards this is to be able to compare the vulnerability of components in a computing platform. In this section we analyze the information leakage due to sequential prefetching algorithm; a prefetching algorithm in the cache memory.

The misses in cache memory have considerable overheads in the performance of a program. In order to reduce these overheads many modern processors have hardware that automatically prefetches data into the cache. The algorithms work by predicting the memory locations accessed by a program. A commonly used prediction algorithm called *sequential prefetching*, monitors memory accesses made and prefetches subsequent blocks. Sequential prefetching is presented in Algorithm 7.2. If t_i is a memory address and t_{i+1} its subsequent address location, an access to t_i would automatically prefetch t_{i+1}.

Each point in the timing profile ($\mathcal{D}_i[j]$) corresponds to the deviation in average execution time when a plaintext byte x_i is kept constant. Owing to this constant, a specific block in the look-up table (corresponding to the index $x_i \oplus k_i$) is accessed in every encryption that is considered in the average. Since all other inputs to the cipher are random, and the randomness diffuses to every round, the other accesses to the look-up tables are at random locations. With respect to the number of cache misses that occur due to the look-up table, probability of a cache miss in the memory block corresponding to the index $x_i \oplus k_i$ is one assuming that there are no conflict misses.

The probability of cache misses in all other memory blocks in the look-up table is less than one, depending on wheather the block was accessed during the encryption. In a cache memory that does not support prefetching, all these memory blocks have an equal probability of being loaded into the cache. However, when prefetching is present, the uniformity in the probability of a cache miss across all memory blocks of the table is not true. Depending on the prefetching algorithm used, the probability of obtaining a cache miss may vary from one memory block of the table to another. We explain the behavior with respect to the sequential prefetching algorithm. We initially assume that the table is isolated, which means that no memory access outside the table can prefetch a memory block of the table. Later we will relax this assumption.

With respect to the sequential prefetcher in Algorithm 7.2, the memory blocks of the table can be split into three categories (depicted in Fig. 7.6).

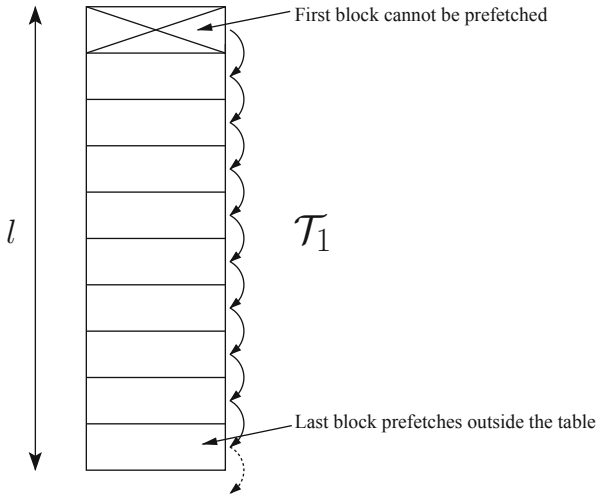


Fig. 7.6 Prefetchable blocks of a table used in a block cipher

- The first memory block of the table can never get prefetched because it would require a memory access to a block preceding the first block. This is outside the table boundary, therefore by our assumption, cannot prefetch any part of the table.
- All blocks of the table except the last, prefetch a memory block inside the table (a valid prefetching by our assumption).
- An access to the last block of the table prefetches a memory block outside the table boundary.

Since the number of cache misses that occur during the encryption have a significant impact on the execution time, this difference in memory blocks results in nonconstant time executions of the cipher and a timing profile which leaks information. In the following discussion we understand the impact of this on the timing profile.

Assuming that the table occupies l memory blocks, and no conflict misses arise, one of three scenarios occur depending on which memory block is accessed by the fixed $x_i \oplus k_i$.

- If the first memory block in the table is accessed by $x_i \oplus k_i$, then the second block of the table is prefetched. Assuming a clean cache at the start of encryption, there are $l - 2$ memory blocks remaining, which can result in cache misses. All these $l - 2$ memory blocks are also prefetchable.
- If the last memory block in the table is accessed by $x_i \oplus k_i$, the prefetching is done outside the table and will not affect the table accesses. In this case there are $l - 1$ cache misses that can still occur due to the look-up table and all these memory blocks can be prefetched except for the first.
- If any other memory block other than the first and last is fixed by $x_i \oplus k_i$, then $l - 2$ cache misses can occur and $l - 3$ of these blocks can be prefetched.

These observations are summarized in Table 7.3 along with the average timing expected. When the first memory block is the invariant, the fastest execution is achieved

Table 7.3 Cache misses in a table of size l used in a block cipher with sequential prefetching enabled

Memory block of table fixed	Number of cache misses possible	Number of sequential prefetching possible	Remarks on average time
First	$l-2$	$l-2$	t_1
Any except first and last	$l-2$	$l-3$	t_2
Last	$l-1$	$l-2$	t_3

Comparison of the average timings : $t_1 < t_2 < t_3$

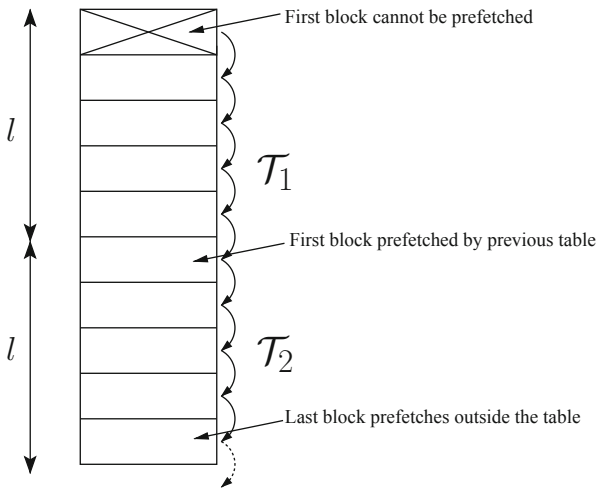


Fig. 7.7 Prefetchable blocks of two adjacent tables used in a block cipher

as it has the least number of cache misses possible and all memory blocks can be prefetched. When the last memory block is the invariant, the slowest encryptions occur because it has the most number of cache misses. Thus, prefetching causes nonconstant encryption time to occur, which is captured in the timing profile.

Relaxing the Assumptions: The previous analysis was made under the assumption that the table is isolated and other memory accesses made by the program do not affect the prefetching in the table. In reality, however, this assumption would not always hold. To show the effect we consider a block cipher implemented with two tables $\mathcal{T}_1$ and $\mathcal{T}_2$ placed side-by-side in memory (as shown in Fig. 7.7). If $x_i \oplus k_i$ accesses $\mathcal{T}_1$, then the last memory block prefetches the first memory block of $\mathcal{T}_2$. Conversely, if $x_i \oplus k_i$ accesses $\mathcal{T}_2$, then the first memory block can be prefetched by the last memory block of $\mathcal{T}_1$. This can cause a reduction in the leakage as there are fewer variations in the encryption time.

7.5 Conclusion

If an adversary is capable of learning the execution behavior of a cipher then she can mount profiled attacks. These are typically the most powerful form of side-channel attacks. The profiled time-driven cache attacks discussed in this chapter is one such profiled attack. Leakage in profiled time-driven cache attacks are due to several causes such as conflict misses, hardware prefetching, and differential timing characteristics of memory loads. The leakage is also significantly affected by the cipher implementation. This chapter analyzed information leakage in hardware prefetching and showed how the information leaked in these attacks can be quantified.

References

1. Neve M, Seifert J-P, Wang Z (2006) A refined look at Bernstein's AES side-channel analysis. In: Lin FC, Lee D-T, Lin B-S, Shieh S, Jajodia S (eds) ASIACCS. ACM, New York, p 369
2. Neve M, Seifert J-P, Wang Z (2006) Cache time-behavior analysis on AES, http://citeseerx.ist.psu.edu/viewdoc/summary?doi=10.1.1.88.4091. Accessed Dec 2014
3. Gueron S (2010) Intel Advanced Encryption Standard (AES) instructions set (Rev : 3.0)
4. Bernstein DJ (2005) Cache-timing Attacks on AES. Tech. Rep.
5. Rebeiro C, Mukhopadhyay C (2012) Boosting profiled cache timing attacks with apriori analysis. IEEE Trans Inf Forensics Security 7(6):1900–1905
6. Bhattacharya S, Rebeiro C, Mukhopadhyay D (2012) Hardware prefetchers leak: a revisit of SVF for cache-timing attacks. In the Workshop Proceedings of the 45-th IEEE/ACM International Symposium on Microarchitecture, MICRO 2012 Workshops, Vancouver, BC, Canada, pp 17–23, IEEE

Chapter 8
Access-Driven Cache Attacks on Block Ciphers

Cache memories are a shared resource in a system. If a process running in a system fetches a block of data into the cache, the data will remain in the cache memory unless evicted. The eviction can be done by any process in the system and can result in a covert timing channel. The channel was first discovered by Wray in [1] and Hu in [2]. Attacks on cryptographic algorithms using the shared feature in cache memories were suggested independently by Percival in [3] and Osvik, Shamir, and Tromer in 2005 [4]. These attacks are known as *access-driven* attacks and use time as a side channel instead of power or electromagnetic traces. We briefly describe selected works in this category of cache attacks.

8.1 Access-Driven Attacks on Block Ciphers

Consider the T-table implementation for AES (Sect. 2.2.1.1). For the first round, the table $\mathcal{T}_j$ is accessed at index $s^{(0)}_{i+j} = (x_{i+j} \oplus k^{(0)}_{i+j})$ for $0 \leq j \leq 3$ and $i \in \{0, 4, 8, 12\}$, where x_{i+j} is an input byte and $k^{(0)}_{i+j}$ the corresponding whitening key byte. Suppose the key byte $k^{(0)}_{i+j}$ is guessed (the guess represented by $\tilde{k}^{(0)}_{i+j}$), then a corresponding index in table $\mathcal{T}_j$ can be computed as $\tilde{s}^{(0)}_{i+j} = x_{i+j} \oplus \tilde{k}^{(0)}_{i+j}$. If the guessed key is correct (i.e., $k^{(0)}_{i+j} = \tilde{k}^{(0)}_{i+j}$) then the computed index is also correct (i.e., $s^{(0)}_{i+j} = \tilde{s}^{(0)}_{i+j}$) and vice versa.

In [4–6], Osvik, Tromer, and Shamir showed two methods that an adversary monitoring the covert timing channels in cache memories can use to identify when the following equality $\langle s^{(0)}_{i+j} \rangle = \langle \tilde{s}^{(0)}_{i+j} \rangle$ holds. The methods *evict+time* and *prime+probe* require that the adversary knows (or learns) the cache sets occupied by the table $\mathcal{T}_j$. We summarize the methods below:

Evict + Time: For every possible value of the table index ($\langle \tilde{s}^{(0)}_{i+j} \rangle$) perform the following operations:

1. Trigger an encryption (Algorithm 2.1) for a random input $\mathbf{x}$.
2. For a guess of $\tilde{s}^{(0)}_{i+j}$, determine the memory address of $\mathcal{T}_j[\tilde{s}^{(0)}_{i+j}]$ and compute the cache set it gets mapped into. Denote this cache set as M.

C. Rebeiro et al., *Timing Channels in Cryptography,*
DOI 10.1007/978-3-319-12370-7_8

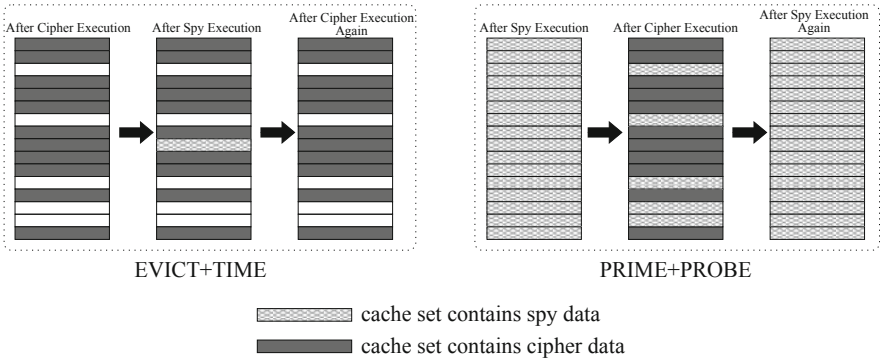


Fig. 8.1 Cache sets corresponding to table $\mathcal{T}_j$ for evict+time and prime+probe access driven attacks

3. Perform operations so that all data present in cache set M due to the first encryption is evicted.
4. Trigger a second encryption with $\mathbf{x}$ again and time it.

Figure 8.1 shows the various stages of the cache sets used by the table $\mathcal{T}_j$. There are two possible outcomes after the second encryption:

- If $\langle s_{i+j}^{(0)} \rangle = \langle \tilde{s}_{i+j}^{(0)} \rangle$ implying $(\langle k_{i+j}^{(0)} \rangle = \langle \tilde{k}_{i+j}^{(0)} \rangle)$, then the cache set M is always accessed by both encryptions (in steps 1 and 4). Particularly, the second encryption would *always* (with probability $= 1$) result in a cache miss for the access $\mathcal{T}_j[s_{i+j}^{(0)}]$. We call such a miss as an *ever-present* cache miss.
- If $\langle s_{i+j}^{(0)} \rangle \neq \langle \tilde{s}_{i+j}^{(0)} \rangle$ implying $(\langle k_{i+j}^{(0)} \rangle \neq \langle \tilde{k}_{i+j}^{(0)} \rangle)$, then for a value of $\mathbf{x}$, the cache set M is not accessed at $\mathcal{T}_j[\tilde{s}_{i+j}^{(0)}]$. However, the set *may* be accessed by the other memory accesses to $\mathcal{T}_j$. Thus for the second encryption (in step 4), the probability of a cache miss is ≤ 1 for the access $\mathcal{T}_j[s_{i+j}^{(0)}]$.

If several values of $\mathbf{x}$ are considered, the ever-present cache miss at $\mathcal{T}_j[s_{i+j}^{(0)}]$ when $\langle s_{i+j}^{(0)} \rangle = \langle \tilde{s}_{i+j}^{(0)} \rangle$, would cause a slight increase in the expected encryption time compared to when $\langle s_{i+j}^{(0)} \rangle \neq \langle \tilde{s}_{i+j}^{(0)} \rangle$ where the cache miss at $\mathcal{T}_j[s_{i+j}^{(0)}]$ is not always present. This difference in time can be detected by an adversary to determine the correct value of $\langle s_{i+j}^{(0)} \rangle$ there by obtaining $\langle k_{i+j}^{(0)} \rangle$ (from the known values of the plaintext).

In [7], Bogdanov, Eisenbarth, and Paar use the Evict+Time strategy to mount a differential cache-collision attack on AES. The main idea of the attack is to choose pairs of plaintexts such that they cause *wide collisions*. Wide collisions in AES are explained using Fig. 8.2. Plaintexts P_1 and P_2 are chosen randomly as inputs to AES in such a way that

- Values on the main diagonals $A = \{a_i\}$ and $E = \{e_i\}$, $0 \leq i \leq 4$ are chosen randomly and independently of each other such that $\exists i, a_i \neq e_i$.
- The remaining bytes of P_1 and P_2 are chosen randomly and are made pairwise equal.

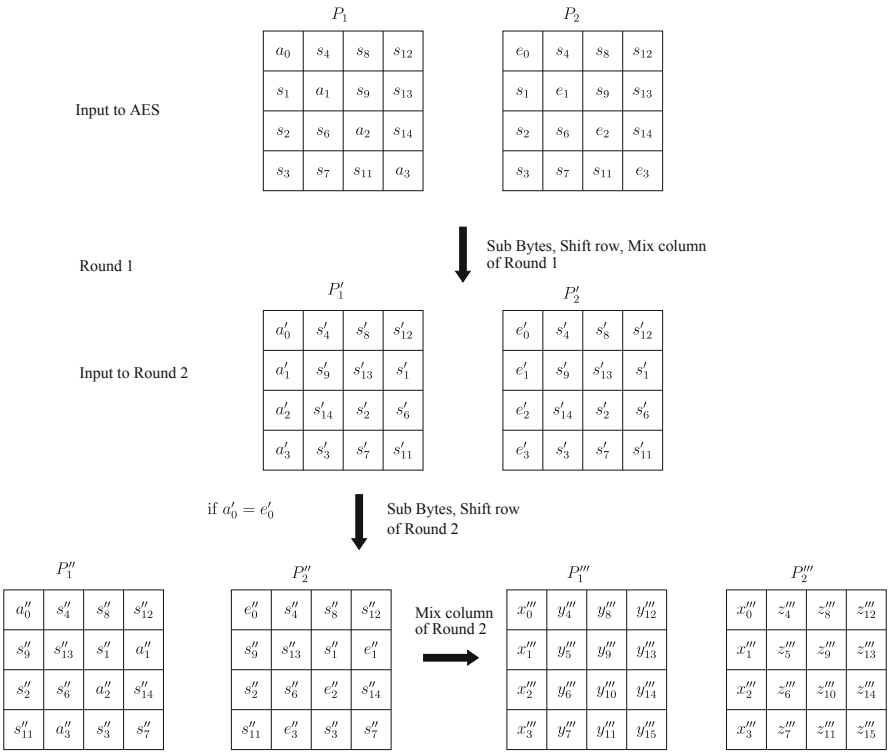


Fig. 8.2 Wide collisions in AES

From Fig. 8.2, P'_1 and P'_2 are the outputs of the first round. It is observed that in both of them, only the bytes in the first column varies. The other bytes are the same. If we trace the inputs to the third round, the values of first column of P'''_1 and P'''_2 are pairwise equal which leads to five AES s-box operations (one in the second round and four in the third), which possess either pair wisely equal or pair wisely distinct values.

Wide collision can only be observed when $a'_0 = e'_0$. The evict+time technique is used to identify plaintext pairs with wide collisions. An existence of a wide collision would cause on average the second encryption to be faster by a margin of five compared to when there is no wide collision. The wide collisions are then used to construct a set of four nonlinear equations, which when solved reveal bytes of the key.

Prime + Probe: In the evict+time method, the time for the memory access to $\mathcal{T}_j[s^{(0)}_{i+j}]$ (where $0 \leq j \leq 3$ and $i \in \{0, 4, 8, 12\}$) gets reflected in the total execution time of the cipher, thus leading to the attack. However, this technique delivers a low success due to the presence of additional memory accesses and other code that executes during the encryption. Further, there is considerable noise from sources such as instruction scheduling, conditional branches, and cache contention thus resulting in a low signal-to-noise ratio (SNR). In the prime+probe method, smaller

codes are timed thus leading to a attacks that are more successful. The steps involved in the prime+probe is as follows:

1. Define an array A as large as the cache memory and read a value of A for every memory block (thus filling the entire cache with A).
2. Trigger an encryption with a random input $\mathbf{x}$.
3. For a guess of $\langle \tilde{s}_{i+j}^{(0)} \rangle$, determine the cache set that $\mathcal{T}_j[\langle \tilde{s}_{i+j}^{(0)} \rangle]$ gets mapped into. Denote this cache set as M.
4. Access A at indices which get mapped to the cache set M and time the individual accesses. Figure 8.1 shows the various stages of the cache sets used by the table $\mathcal{T}_j$.

If $\langle \tilde{s}_{i+j}^{(0)} \rangle$ is correct (i.e., $\langle \tilde{s}_{i+j}^{(0)} \rangle = \langle s_{i+j}^{(0)} \rangle$), then $A's$ data present in the cache set M would be evicted with probability $= 1$ during the encryption in step 2. However, this probability can be less than one if $\langle \tilde{s}_{i+j}^{(0)} \rangle$ is incorrect. Further, an eviction of $A's$ data at M would result in a cache miss in the fourth step; identified by a longer memory access time. However, if $A's$ data at M is not evicted, then the last step would have a cache hit therefore a shorter access time. The correct $\langle \tilde{s}_{i+j}^{(0)} \rangle$ (thus $\langle \tilde{k}_i^{(0)} \rangle$, again assuming that the plaintext is known) can therefore be identified by repeating the four steps with several inputs.

8.1.1 Second Round Access-Driven Attack on AES

For a table with $l \cdot 2^{\delta}$ elements, each element has an entropy $n = \delta + \log_2 l$, where 2^{δ} is the number of elements in the table sharing a memory block and l is the number of memory blocks occupied by the table. The first round attack can reveal at most $\log_2 l$ bits of each key byte. In order to reveal the entire key byte, the second round of AES needs to be targeted [4–6].

Consider the first table access in the second round. From Algorithm 2.1, Fig. 2.3, and the key scheduling algorithm of AES, this is

$$s_0^{(1)} = 2 \cdot S[x_0 \oplus k_0^{(0)}] \oplus 3 \cdot S[x_5 \oplus k_5^{(0)}] \oplus S[x_{10} \oplus k_{10}^{(0)}] \oplus S[x_{15} \oplus k_{15}^{(0)}] \\ \oplus k_0^{(0)} \oplus S(k_{13}^{(0)}) \oplus 1 \,. \tag{8.1}$$

The value of $\langle s_0^{(1)} \rangle$ is affected by keys $k_0^{(0)}$, $k_5^{(0)}$, $k_{10}^{(0)}$, $k_{13}^{(0)}$, and $k_{15}^{(0)}$ each occupying one byte. The first round attack reveals $\log_2 l$ bits of each of these key bytes leaving δ bits unknown, Thus there is a space of $5 \cdot 2^{\delta}$ keys that need to be searched. The evict+time or prime+probe methods can be used along with Eq. 8.1 to identify the correct key from this key space.

8.1.2 A Last Round Access-Driven Attack on AES

As described in Sect. 2.2.1.1, the last round of some implementations of AES use a different table (called $\mathcal{T}_4$). This table is exclusively used in the last round and

can lead to ciphertext only attacks. A byte of the ciphertext can be expressed as $y_i = k_i^{(10)} \oplus S(s_i^{(9)})$ where $0 \leq i \leq 15$ (here we have ignored the last round ShiftRows operation as it does not affect the attack complexity). Thus $k_i^{(10)} = y_i \oplus S(s_i^{(9)})$. Evict+time or prime+probe methods can be used to determine $\langle S(s_i^{(9)}) \rangle$, and therefore $\langle k_i^{(10)} \rangle$ can be determined.

Neve and Seifert in [8] developed two ways to determine all bits of $k_i^{(10)}$. The first method called the *nonelimination method*, is based on the fact that for a given value of y_i, the input to the SubBytes in the last round (i.e., $s_i^{(9)}$) is fixed. This implies that the cache set accessed for a given y_i is always the same. Now consider that $y_i = t_1$ corresponds to $s_i^{(9)} = o_1$, where $0 \leq t_1, o_1 \leq 255$. Similarly $y_i = t_2$ corresponds to $s_i^{(9)} = o_2$, where $0 \leq t_2, o_2 \leq 255$. Thus,

$$\begin{aligned} t_1 &= k_i^{(10)} \oplus S(o_1) \\ t_2 &= k_i^{(10)} \oplus S(o_2) \end{aligned} \tag{8.2}$$

The access attack reveals $\langle o_1 \rangle$ and $\langle o_2 \rangle$. This leaves 2^δ possible key options for $k_i^{(10)}$. This key space can then be reduced by using the relation $t_1 \oplus t_2 = S(o_1) \oplus S(o_2)$ obtained from Eq. 8.2. Further reduction of key space is possible by considering different pairs of values for y_i. The authors in [8] estimate that 186 pairs are required to uniquely identify the correct key.

The second method is based on eliminating incorrect values of $s_i^{(9)}$. It uses the fact that the table $\mathcal{T}_4$ is accessed only 16 times and not all memory blocks of the table are likely to be accessed during the encryption. The memory blocks not accessed can be detected by the evict+time or the prime+probe methods. It is certain that $S(s_i^{(9)})$ is not one of the values in the unaccessed blocks. Thus if n blocks of $\mathcal{T}_4$ are not accessed during the encryption, then $n \cdot 2^\delta$ values of $s_i^{(9)}$ can be eliminated. By using different inputs to the cipher, more values of $s_i^{(9)}$ can be eliminated until a unique value is obtained. This corresponds to a unique value of $k_i^{(10)}$.

8.2 Asynchronous Access-Driven Attacks

The access-driven attacks discussed so far were synchronous, wherein it is assumed that the adversary can trigger an encryption. A less restrictive form of the attack is the asynchronous mode, where a nonprivileged adversary running concurrently in an independent user space, needs only to monitor the cache activity to determine cache sets accessed by the executing cipher. No triggering of encryption is required. Instead, the only requirements is that the adversary executes in the same processor and concurrently with the cipher. Asynchronous access driven attacks were pioneered by Percival in [3], who demonstrated an attack on RSA. Osvik et al. and Neve then explored the feasibility of asynchronous attacks on AES [4–6, 8], though the practicality of the work remains unclear. Aciiçmez et al. [9] were the first to present a

practical asynchronous access-driven using instruction caches on OpenSSL's DSA, while [10] demonstrated the first practical asynchronous attack on AES.

In an asynchronous attack, the adversary runs a spy process which periodically reads or writes data into cache lines and monitors the time it takes. An eviction of the spy's data by another process (such as an executing cipher) from the cache will result in a longer memory access time due to the cache miss that occurs. This gives clues about the memory access patterns of the cipher process. Such leakages are also possible when the spy and cipher are executing in different security zones. For instance, in cloud computing virtualized environments [11], several users share the same hardware, albeit in different security zones.

A system vulnerable to such asynchronous attacks utilizes at least one of the following features in the computer system: symmetrical multithreading [3; 6] or preemptive operating system (OS) scheduler [8; 10].

Symmetrical multithreading allows multiple processes to execute simultaneously on a single processor. The hardware resources including the cache memories are shared between the processes. The cache memory acts as a covert channel transmitting information about the memory access patterns of the cipher.

Preemptive OS Scheduling divides CPU time into equally spaced intervals called *slices*. At the beginning of a slice, a process gets allocated to the CPU and executes until it voluntarily relinquishes the CPU or the time slice completes. A new process may then get allocated to the CPU by a process known as *context switching*. In [8], Neve and Seifert suggest that spy processes can exploit such schedulers to obtain covert information about a cipher's execution, though no explicit details about the construction were given. In [12], Tsafrir, Etsion, and Feitelson present a practical malicious code that can exploit context switching in OS schedulers. The intuition is that the malicious code starts executing at the beginning of a slice, but yields the processor before the slice completes. Another process is then scheduled for the remaining time interval in the slice. The authors show several applications of the malicious code such as denial of services, bypassing profiling and administrative polices, etc. Bangerter, Gullasch, and Krenn use such a malicious code to develop a fine grained access-driven cache timing attack in [10]. In the next section we show how the OS scheduler can be used monopolize the system. The find grained attack on AES is then described.

8.3 Secretly Monopolizing the CPU scheduler

Computer systems are governed by two clocks—*hardware clocks* drive the instruction cycle while the *operating system clocks* control the system activity, measure time passage, provide timing services, and maintain control. Unlike the hardware clock, whose resolution is determined by the processor frequency (and hence cannot be tuned), the system clock frequency is determined by the OS at boot-up.

Clock resolution in the OS clock has a strong influence on scheduling. In [12], it is shown how these clocks can be tricked to allow a malicious code to secretly consume a large percentage of the CPU cycles without any accounting. The phenomenon is called "cheat" attack and exploits two paradigms of some OS schedulers.

- CPU usage is accounted by periodic sampling interrupts.
- Processes that use less CPU have higher priority.

A periodic hardware clock interrupt termed as *tick* is used for measuring CPU usage. This accounting information is mostly used by priority-based schedulers for calculating priority of processes. When a tick occurs, the process executing is billed for the entire interval since the previous tick. The schedulers rely on this periodic sampling interrupts of a low-precision clock to account for the CPU usage. Due to this coarse granularity of the periodic sampling the billing may not be accurate, thus leading to the inaccurate information being used by the scheduler. This inaccuracy is exploited in the *cheat* attack procedure.

The exact time for which the process is scheduled to run is the quantum of a process. The effective quantum can have widely varying values and may or may not be greater than a clock tick, though CPU usage is accounted only at the clock ticks. Thus when a clock tick occurs the running process gets billed in integral multiples of the periodic sampling interval. If a quantum is shorter than the tick duration, then the process has higher probability of not getting billed. So for processes having intervals of quanta shorter than a clock tick, many smaller quanta can consume arbitrarily small CPU cycles remaining unbilled, while if a short quanta includes a clock tick then it gets overbilled by the entire tick duration though actually consuming a small fraction of the periodic interval. Similarly, the situation is true for bigger quanta. Since the probability that a quantum includes a tick is proportional to its duration, on average the overbilling and underbilling effect tends to cancel out to provide reasonably accurate billing.

If a process maliciously manages to exploit this coarse granularity phenomenon for accounting the CPU usage then it never gets billed. The cheat process thus performs the following:

- Schedules to run just after the tick occurs.
- Finishes its work within the fraction of the tick duration.

These are repeated over several ticks. Thus the cheat process makes sure that it is never scheduled when the periodic tick occurs. This work demonstrates the working principles of a cheat program in which a process at user level is not only capable of consuming any percentage of the CPU but also the accounting of this extra CPU usage does not get reflected in the monitoring tools since other processes gets billed instead of the consuming process.

As shown in Fig. 8.3, when a tick occurs, the OS performs billing and allows the pending process having highest priority to execute in the CPU. Thus the cheat process can easily take control over the CPU just after a clock tick occurs. In order to avoid billing, if the cheat process is capable of preempting itself using a fine-grained timer of high frequency before the next tick occurs, then it successfully prevents itself

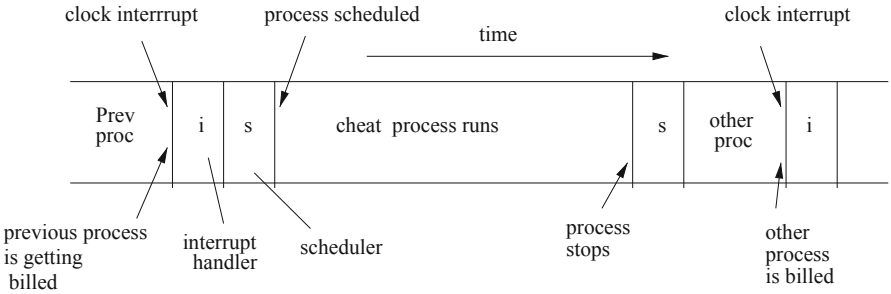


Fig. 8.3 Philosophy of the cheat program

from getting billed. Thus the OS is forced to believe that the malicious process is consistently sleeping. This illusion increases the priority of the malicious code from the OS perspective. The policy for the recent schedulers is such that it always assign a higher priority to processes that have lower CPU usage and thus their requests of consuming CPU cycles are serviced first. Thus for a malicious process which is accounted as sleeping, it is easy to be fired just after a periodic clock interrupt occurs.

In order to prevent itself from getting billed the process constructs a fine-grained timing measurement mechanism by reading the values of a timestamp counter. The timestamp counter is read using the "rdtsc" assembly call, which returns the value of the hardware timestamp counter. This assembly call can be made from the user level and returns the timestamp of the system in clock cycles.

Listing 8.1 illustrates a C code implementation of the cheat program. The subroutine *get_cycles* enables the user to read the timestamp counter value using a *rdtsc* assembly call. The assembly call is explained in details in Sect. 4.2.1. The interval between two consecutive ticks is determined using subroutine *cycles_per_tick*. The interval is observed for 1000 such measurements and is averaged to obtain a precise value of the number of clock cycles per tick. A zero sleep is inserted in the function which wakes up the process at the next clock interrupt.

These subroutines are actually used by the *cheat_attack* which successfully steals a fraction of CPU cycles without getting billed. The cheat process continuously polls to see that whether it has consumed more that the fraction it actually wants to consume. The attack code continuously iterates over the *while* loop and checks whether the desired fraction is over. When the counter crosses the limit, the process blocks itself till the next tick by calling nanosleep function with zero time. While it polls to check for the desired interval, additionally it executes some short work side-by-side. Thus when the desired fraction of the periodic interrupt has been used by the *cheat_attack* it preempts itself and allows other processes to run.

Listing 8.1 Simple C implementation of cheat program

```
#define _POSIX_C_SOURCE 199309
#include <stdio.h>
#include <stdlib.h>
#include <sys/time.h>
struct timespec zero = {0,0};  // sleep for zero ns before
                               // the next tick
typedef unsigned int cycle_t; // unsigned 32 bit integer to store
                               // the clock cycles

// returns the value of the timestamp
// counter in number of clock cycles
inline cycle_t get_cycles()
{
  cycle_t ret;
  asm volatile("rdtsc" : "=A" (ret));
  return ret;
}

//calculating the number of clock cycles per clock tick
cycle_t cycles_per_tick()
{
  int i;
  nanosleep(&zero,0);     // sync with tick
  cycle_t start = get_cycles();
  for(i=0 ; i<1000 ; i++)    // measures cycles for 1000 ticks
    nanosleep(&zero,0);
  return (get_cycles() - start)/1000;   // measure average cycles
                                          // between clock ticks
}

//Subroutine for the Attack
void cheat_attack( double fraction )
{
  cycle_t work, tick_start, now;
  work = fraction * cycles_per_tick();    // clock cycles to be
                                          // consumed within a
                                              tick duration
  nanosleep(&zero,0);               // sync with tick
  tick_start = get_cycles();        // timestamp counter at the
                                      // start of tick
  // poll continuously to check
        // whether the desired fraction is consumed
  while( 1 ) {
    now = get_cycles();
    if( now - tick_start >= work ) {
      nanosleep(&zero,0); // avoid bill
      tick_start = get_cycles();
    }
    // do some short work here...
    system("gcc test.c -o t1"); // compiling a test process
    system("./t1"); // calling a test process to run

  }

}


int main()
{
  cheat_attack(0.95);      // calling the cheat attack with the
                                // desired fraction
}
```

But the above implementation has certain drawbacks as in the cheat process polls continuously to check whether the desired interval has expired. This continuous polling leads to a significant overhead for the cheat process. The following subsection provides a brief discussion on a server–client setup which shows a significant improvement upon the previous strategy.

8.3.1 Cheat Server

The cheat server is a machine with which the client machine (in which the cheat code executes) communicates, such that the cheat process blocks itself on receiving a predefined message from the server. This communication with the cheat server can be viewed as a substitute to the finer grained timing measurements required to preempt the cheat process successfully.

The communication between the cheat process and the network server is carried out through a predefined protocol named as cheat protocol. The protocol is explained briefly as follows: Initially the cheater process allows the cheat client to establish a connection with the remote cheat server. After the client machine establishes the connection, the client sends a packet to the cheat server requesting the server to send it a message after a fine-grained time interval which is essentially less than the interval to next periodic interrupt of the client. This request is sent via a User Datagram Protocol (UDP) data packet which contains the desired interval in nanoseconds. The server on receiving a request, waits for the interval to expire with a precise granularity and communicates with the client as soon as the timer exceeds the interval. Thus, there can be two possibilities: either the server is having an OS of higher tick rate or the server busy waits on a cycle counter for the request time interval to signal the client precisely. The client on the other hand allows the target application to run in the local processor while it polls the network for the message to arrive. As soon as it receives the message, in order to prevent itself from getting billed the client goes to nanosleep with zero offset. This essentially prevents the target application from getting billed, on addition to this due to zero-sleep, the application gets rescheduled on the immediate next tick. The cheat client again performs the same sequence of communication on every tick.

The immediate advantages of this cheat server above the previous implementation is that since the cheat server is dedicated to provide the reminder to go for sleep, the client system thus can sleep wait until it receives a message while the target application runs on the local host unmodified. Since the cheat code avoids the polling, the loss in throughput is prevented to a great extent.

But the communication of the cheat client with the server can be detected if the network traffic is monitored. In addition to this there is a serious drawback regarding the delay caused by the network latency. The success of the procedure truly relies on the performance of the server. The next subsection provides an alternative to this procedure and is free from the drawbacks mentioned above.

8.3.2 Binary Instrumentation

Binary instrumentation is an easier alternative for implementation of the cheating code and by incorporating it, any benign code can be converted into a cheater code. The cheat analysis subroutine is listed in Listing 8.2. Any arbitrary program can include this subroutine and if this routine is invoked a number of times then essentially it turns into a cheat program.

Listing 8.2 Cheat analysis routine

```
void cheat_analysis()
{
  cycle_t c = get_cycles();

  if(c - tick_start >= work)
  {
    nanosleep(&zero, 0);

    tick_start = get_cycles();
  }
}
```

Due to the two features—accounting the CPU usage and system timer services being done at periodic ticks in general purpose systems any arbitrary user level code can systematically sleep at the periodic ticks monopolizing the CPU.

The following section provides an attack strategy which exploits this fine grained scheduling to practice a *denial of service* attack. The attack falls in the category of Access driven attacks and is demonstrated using timing as side channel on AES block cipher.

8.4 Fine Grained Access-Driven Attack on AES

The previous access-driven cache-timing attacks on AES [4-6, 8] in Sect. 8.1 and 8.2, used the fact that the spy determines all the cache sets accessed by AES only after the encryption is complete. Due to the large granularity of measurements in the attacks, it was not possible for the spy to determine intermediate accesses done by AES. In [10], Bangerter, Gullasch, and Krenn show a way of exploiting the OS scheduler to obtain information about every single cache access performed by AES.

In order to practically mount the attack, the completely fair scheduler (CFS), which is typically used in the Linux kernels, is exploited. About 100 spy threads are launched in the system. When a spy thread gets scheduled into the processor, it first makes the timing measurements of cache accesses, and then waits until most of the time slice is complete before blocking itself. The cipher then executes in the small time interval that remains in the time slice. Typically this attack features the *denial of service* to the cipher process, exploiting the underlying scheduler policy.

The cipher gets scheduled for a very small interval of time before being preempted by the scheduler.

This interval is just sufficient for the cipher to make one memory access. The next spy thread then gets scheduled into the processor and the single access made by the cipher is determined by measuring the memory access time to access its data. In this way, the cipher is executed very slowly, and each memory access it makes can be tracked by the spy.

Bangerter et al. [10] demonstrated the first practical asynchronous attack on AES. In this work the OS scheduler vulnerabilities are exploited to perform a fine grained access driven cache attack on practical implementation of AES. A malicious code is executed with multithreading capabilities exploiting the CFS policies in order to obtain cache access patterns of running encryption. Thus this attack does not require any direct synchronization with the victim process though exploits the timing as the source of information to decide whether an intermediate cache access is a hit or miss.

8.5 Attack Procedure

8.5.1 Completely Fair Scheduler

Any general purpose multitasking OS provides an illusion to the user to run several parallel processes at a time. The scheduler is responsible for the multiplexing between processes and threads. Recent Linux systems are equipped with CFS, which justifies its name by asymptotically behaving like an ideal system. Virtual run times (τ_is) are associated with every running process P_i and to ensure complete fairness of the scheme, the virtual run times of all executables would increase simultaneously. But in real systems this phenomenon of simultaneous increment in the virtual run time is not possible since at any point of time, a single process takes hold of the CPU and thus run time of that particular running process gets modified.

For real systems, the scheduler maintains a time line of the respective virtual run times of processes that are ready to run. Consider a queue of processes ordered in increasing sequence from left to right. The leftmost entry in the sequence is the process that is least favored since it has been waiting for greatest interval while the process on the rightmost in the ordered sequence is the most favored process because it has been serviced recently. The scheduler defines *unfairness* as the difference in the virtual run times of the rightmost and the leftmost entries in the sequence as $\Delta\tau = \tau_{right} - \tau_{left}$ and the scheduler ensures fairness by bounding the unfairness value ($\Delta\tau$) at any point of time to be lesser than a predefined maximum unfairness ($\Delta\tau_{max}$). The policy of CFS is such that the scheduler preempts the rightmost process as soon as its virtual run time exceeds the maximum unfairness value and in turn activates the leftmost process in the time line. This process is repeated to provide all running processes a fair share of CPU.

Similar to the running processes, if a process blocks itself at virtual run time τ_{block}, the virtual run time of the process on unblocking ($\tau_{unblock}$) itself must be initialized to a value such that it is serviced as soon as possible. So initialization of $\tau_{unblock}$ is made

such that the unfairness amount $\Delta\tau$ exceed $\Delta\tau_{max}$. Thus the scheduler immediately preempts the running process and allows the unblocked process to perform the events for which it was waiting. The *sleeper fairness* criteria of the scheduler says that if the process sleeps for a significant amount of time such that $\tau_{block} < \tau_{right} - \Delta\tau_{max}$, then the blocking process on being unblocked is initialized as $\tau_{unblock} \leq \tau_{right} - \Delta\tau_{max}$ and the process eventually occupies the leftmost position in the time line. For the generic case, the scheduler policy maintains that the virtual run time for an unblocking process is initialized to a value ($\tau_{unblock}$) which is greater than or equal to the virtual run time when it blocked itself (τ_{block}). Thus, $\tau_{unblock} = max(\tau_{block}, \tau_{right} - \Delta\tau_{max})$.

8.5.2 Denial of Service Exploiting Completely Fair Scheduler

Bangerter et al. in [10] exploits the sleeper fairness of the CFS to surface an attack on the encryption process. The malicious attacker observes the timing measurements of cache accesses via a multithreaded spy process $\mathcal{S}$. The spy process $\mathcal{S}$ launches large number of threads and the blocking or unblocking of the subsequent threads are scheduled sequentially in a way that the victim process $\mathcal{V}$ gets scheduled between two subsequent thread scheduling. The interval for which the victim process runs is sufficient for making one cache access before it is pre-empted due to the unblocking of the subsequent thread. Thus each cache accesses performed by the victim is monitored by the multiple spy threads at fine grained timing interval. The attack principle is detailed as follows:

1. The spy process activates a large number of threads, where initially the virtual run times of all threads are initialized to a significantly low value by blocking itself for sufficient time. After this initialization all the launched threads performs the timing measurements in a round-robin fashion.
2. At any point of time, thread i determines the memory accesses performed by victim process by observing the access times.
3. Spy thread i before blocking itself computes t_{sleep} and t_{wakeup} which are virtual run times in the time line for blocking of the ith thread and the unblocking time for the $(i+1)^{\text{th}}$ thread. Thread i requests a timer to wakeup thread $i+1$ at t_{wakeup} before entering into the busy-wait loop until t_{sleep} is reached.
4. The ith thread blocks itself at t_{sleep} and at this point of time there are no spy threads that are ready to run. The $(i+1)^{\text{th}}$ thread is only activated when t_{wakeup} is reached.
5. The scheduler activates the victim process at t_{sleep}. The victim process can at most run for the interval $t_{wakeup} - t_{sleep}$ since at t_{wakeup} the timer expires and unblocks thread $i+1$. Since the spy threads are activated one after another in order, thread $i+1$ will be serviced immediately at t_{wakeup} since it is the leftmost thread in the time line.
6. Thus, unblocking of the $(i+1)^{\text{th}}$ thread causes a preemption of victim V and the above scenario can be repeated successively for a large number of spy threads in cache access driven attacks exploiting denial of service for the victim process $\mathcal{V}$.

In this work, $t_{wakeup} - t_{sleep}$ is set as 1500 machine cycles, where only 200 cycles are actually being used by the victim to perform the encryption.

8.5.3 Using Timing as Side Channel for Cache Access Attack

Listing 8.3 is a C code implementation for the timing measurements of cache accesses. The spy process can use this implementation to learn the information about the table look-ups only up to the cache line granularity. The reason behind this is that the requested data is loaded from the main memory to the cache memory in data blocks, equal to the size of cache line. So if any data is requested to be loaded from the main memory location, the entire cache line containing the main memory location gets loaded into the cache memory. The spy process measures the time for each look-up table accesses and determines whether any particular access is a hit or miss by looking at the time taken to access the respective locations.

Listing 8.3 Subroutine used by spy process checking the parts of look-up table that have been accessed by other processes

```
#define CACHELINESIZE 64
#define THRESHOLD 200
unsigned measureflush(void *table, size_t tablesize, uint8_t *
    bitmap) {
  size_t i;
  uint32_t t1,t2;
  unsigned bit, n_hits = 0;
  for(i=0; i< tablesize/CACHELINESIZE; i++) {
    _asm_(" xor %%eax, %%eax    \n"
          " cpuid               \n"
          " rdtsc               \n"
          " mov %%eax, %%edi    \n"
          " mov (%%esi), %%ebx  \n"
          " xor %%eax, %%eax    \n"
          " cpuid               \n"
          " rdtsc               \n"
          " clflush (%%esi)     \n" :
          "=a" (t2),
          "=D" (t1) :
          "S" ((const char*) table + CACHELINESIZE * i) :
          "ebx", "ecx", "edx", "cc");
      bit = (t2 - t1 < THRESHOLD) ? 1 : 0;
      n_hits += bit;
      bitmap[i/8] &= ~(1 << (i%8));
      bitmap[i/8] |= bit << (i%8);
  }
  return n_hits;
}
```

The timing measurements for the respective table locations are treated as following:

1. The *rdtsc* as appears in Sect. 4.2.1 instruction is used to observe the timestamp counter values before and after the look-up accesses.
2. The event cache hits and misses can be distinguished through a thresholding scheme where if a timing measurement is less than a threshold then the spy process decides that it is cache hit, otherwise if the measurement exceeds the threshold then the spy concludes the access to be a miss. Rigorous testing on a desired platform will reveal the actual threshold for a system. To the best of our knowledge, there are no well defined techniques to estimate the threshold accurately.
3. A bitmap is maintained so as to keep track at which location cache hits and misses occurred.

The steps are performed on the entire look-up table for AES for each and every cache lines. Since data from the main memory is accessed in cache block, access attacks can only reveal cache accesses in the granularity of cache lines. The timing information is thus used by the attacker to construct a bitmap containing the information of cache hits and misses.

8.5.4 Denoising by Neural Network

Normally the timing measurements as obtained from the explained procedure are highly affected by noise from other processes running concurrently on the system. The events of cache hit and misses for the victim process get hugely affected by the cache accesses from other running processes in the system. This may result in faulty conclusions regarding the key space. Another drawback of this procedure is that the fine-grained timing measurements are not perfect, thus single timing measurements may lead to misleading results.

In this work, Bangerter et al. [10] proposes a denoising technique using the artificial neural network (ANN) which is used efficiently to improve the success of the attack. The inputs to the ANN is bitmap which represents cache hits and misses for the respective cache access across time. The ANN filters out the noise and outputs the probabilities for each memory accesses being actually performed by the victim process.

8.6 Conclusion

The chapter discussed strategies for access-driven timing attacks, which are capable of identifying individual cache accesses made by the cipher. Several attacks on block cipher were discussed. The attacks are immensely benefited by multithreaded CPU environments. Further, by exploiting CPU scheduler policies, every memory access made by the cipher can be determined.

References

1. Wray JC (1991) An analysis of covert timing channels in research in security and privacy. In: Proceedings of the 1991 IEEE Computer Society Symposium, May 1991, pp 2–7
2. Hu W-M (1992) Lattice scheduling and covert channels. In: Proceedings of the 1992 IEEE symposium on security and privacy. SP '92. IEEE Computer Society, Washington DC, pp 52–61
3. Percival C (2005) Cache missing for fun and profit. In: Proceedings of BSDCan 2005
4. Osvik DA, Shamir A, Tromer E (2005) Cache attacks and countermeasures: the case of AES. Cryptology ePrint Archive, Report 2005/271
5. Tromer E, Osvik DA, Shamir A (2010) Efficient cache attacks on AES, and countermeasures. J Cryptol 23(2):37–71
6. Osvik DA, Shamir A, Tromer E (2006) Cache attacks and countermeasures: the case of AES. In: Pointcheval D (ed) CT-RSA. Lecture notes in computer science, vol 3860. Springer, Berlin, pp 1–20
7. Bogdanov A, Eisenbarth T, Paar C, Wienecke M (2010) Differential cache-collision timing attacks on AES with applications to embedded CPUs. In: Pieprzyk J (ed) CT-RSA. Lecture notes in computer science, vol 5985. Springer, Berlin, pp 235–251
8. Neve M, Seifert J-P (2006) Advances on access-driven cache attacks on AES. In: Biham E, Youssef AM (eds) Selected areas in cryptography. Lecture notes in computer science, vol 4356. Springer, Berlin, pp 147–162
9. Aciiçmez O, Schindler W (2008) A vulnerability in RSA implementations due to instruction cache analysis and its demonstration on OpenSSL. In: Malkin T (ed) CT-RSA. Lecture notes in computer science, vol 4964. Springer, Berlin, pp 256–273
10. Gullasch D, Bangerter E, Krenn S (2011) Cache games—bringing access-based cache attacks on AES to practice. In: IEEE symposium on security and privacy. IEEE Computer Society, Washington DC, pp 490–505
11. Ristenpart T, Tromer E, Shacham H, Savage S (2009) Hey, you, get off of my cloud: exploring information leakage in third-party compute clouds. In: Al-Shaer E, Jha S, Keromytis AD (eds) ACM conference on computer and communications security. ACM, New York, pp 199–212
12. Tsafrir D, Etsion Y, Feitelson DG (2007) Secretly monopolizing the CPU without superuser privileges. In: Proceedings of 16th USENIX security symposium. SS'07. USENIX Association, Berkeley, pp 17:1–17:18. http://dl.acm.org/citation.cfm?id=1362903.1362920. Accessed March 2013

Chapter 9
Branch Prediction Attacks

So far we have looked at timing channels that arise due to cache memories in the system. Cache memories leak information about the memory accesses made by a cipher. In this chapter, we look at information leakage due to branch instructions. If a cipher implementation uses a conditional branch that depends on the secret key, then information about the key can leak through the processor's branch predictor. A misprediction causes the execution to take considerably more time compared with predictions that are correct. This variation in execution time is exploited by attackers to determine bits of the secret key. Several timing attacks based on branch prediction have been proposed. Unlike cache attacks, which are mostly effective on block ciphers, branch prediction attacks are applicable only on public key ciphers such as the Rivest, Shamir, and Adleman (RSA) algorithm. This chapter begins with a review of the RSA implementation before discussing various branch prediction attacks.

9.1 Implementation of RSA

RSA performs modular exponentiation to encrypt or decrypt messages. The encryption of message (M) with the public key e is done as follows: $C = M^e \bmod N$. The corresponding decryption is done using the private key d as follows: $M = C^d \bmod N$. For details about RSA, please refer to Sect. 2.5. In this section, we review how the exponentiation is implemented.

9.1.1 Square and Multiply Exponentiation Algorithm

The square and multiply algorithm performs a squaring at each step, while the multiplication operation is performed only for the exponent bits that are set to one. Algorithm 9.1 represents the details. The function returns $S \leftarrow M^d \bmod N$, where the private key d is $(d_0||d_1||d_2||\cdots||d_i||\cdots||d_{n-1})$. Here, d_0 is the most significant bit and d_{n-1} is the least significant bit of d.

C. Rebeiro et al., *Timing Channels in Cryptography,*
DOI 10.1007/978-3-319-12370-7_9

Algorithm 9.1: Binary version of square andmultiply exponentiation algorithm

```
   Input: M
   Output: S
1  begin
2      S ← M
3      for i from 1 to n − 1 do
4          S ← S ∗ S  mod N
5          if d_i = 1 then
6              S ← S ∗ M  mod N
7          end
8      end
9      return S
10 end
```

This algorithm is unbalanced because multiplication is performed only when $d_i = 1$. Common side channels such as power and timing are capable of identifying this conditional execution. Simple power attacks (SPA) and timing attacks exploit this conditional instruction execution retrieving the bits in d that are 1.

9.1.2 Balanced Montgomery Powering Ladder Implementation

A modification to the square and multiply algorithm protects against obvious side-channel leakages. This is called the Montgomery ladder algorithm. Algorithm 9.2 shows this modification.

The Montgomery ladder performs the entire exponentiation by alternatively modifying the values of two variables that depend on the exponent bits. Algorithm 9.2 has both "if" and "else" statements and every iteration has either of the corresponding code executed. Unlike the square and multiply algorithm, here the number of multiplications performed is always a constant and independent of the value of the key. This balanced mode of operation eliminates obvious side-channel leakage.

9.1.3 Montgomery Multiplication

The integer division required for the modular multiplication and squaring in Algorithms 9.1 and 9.2 is the most time consuming. Montgomery multiplication provides a means to avoid this. For two multiplicands a and b, the algorithm computes $(a * b) \bmod N$. There are four steps in the multiplication.

1. First, an integer R is found such that $gcd(R, N) = 1$. If the RSA modulus N is a k-bit number, then R is generally taken to be 2^k. The extended Euclidean algorithm is used to determine the inverse of R and N, denoted R^{-1} and N^{-1} respectively. Thus $R * R^{-1} = 1$ and $N * N^{-1} = 1$. This has to be done only once before any encryption can start.

Algorithm 9.2: Montgomery Ladder Algorithm

```
1  begin
2      R0 ← 1
3      R1 ← M
4      for i from 0 to n − 1 do
5          if di = 0 then
6              R1 ← (R0 ∗ R1) mod N
7              R0 ← (R0 ∗ R0) mod N
8          end
9          else if di = 1 then
10             R0 ← (R0 ∗ R1) mod N
11             R1 ← (R1 ∗ R1) mod N
12         end
13     end
14     return R0
15 end
```

2. R is used to convert the multiplicands into their Montgomery domain as follows:

$$A = (a * R) \bmod N \tag{9.1}$$
$$B = (b * R) \bmod N.$$

3. The next step is to perform the Montgomery multiplication. This is shown in Algorithm 9.3. This involves three multiplications and some less expensive operations such as addition and right shifts (assuming R is a power of 2, division by R is performed by shifting right). The multiplication $S * N^{-1}$ is not too expensive either, because $\bmod R$ implies that only the least significant R bits need to be considered.
4. The inverse Montgomery transformation is then performed to convert the result to an ordinary integer,

$$z = S * R^{-1} \bmod N. \tag{9.2}$$

From the side-channel perspective, there is an extra reduction step in 4th line of Algorithm 9.3. Due to the conditional branch statement, variations in the execution time will occur. These variations are the source of side-channel exploits. The next section demonstrates this leakage through the processor's branch predictor.

9.2 Timing Branch Mispredictions

The branch prediction unit in the processor is responsible for predicting whether a branch in a program is taken or not-taken. If a branch is predicted taken, instructions from the destination address are fetched. On the other hand, a not-taken prediction causes the CPU to continue fetching instructions in a sequential fashion. A prediction

Algorithm 9.3: Montgomery Multiplication Algorithm

```
1 begin
2     S ← A ∗ B
3     S ← (S + (S ∗ N^{-1} mod R) ∗ N)/R
4     if S > N then
5         S ← S − N
6     end
7     return S
8 end
```

that goes wrong is called a *misprediction*. It causes the pipeline to be flushed and instructions fetched from the correct destination. This incurs a significant overhead in superscalar processors. The difference in execution time between a correct and a wrong branch prediction is used to break ciphers such as the RSA. In this section, we demonstrate the timing difference caused by mispredictions.

We take the example of a Montgomery multiplication implemented to return the square of its input. As seen in Algorithm 9.3, the Montgomery multiplication has an extra reduction step which is executed only if $S > N$. We determine the execution time difference caused by the mispredictions that occur due to this condition in the algorithm.

The experiment performs the operation $a^2 \bmod N$ for an input a using the Montgomery multiplication algorithm. Listing 9.1 shows the pseudocode. The modulus N is a 1024-bit integer similar to the modulus used in RSA. Since this involves numbers much bigger than what normal CPU registers can hold, GNU Multiple Precision Arithmetic Library is used to perform the big number arithmetic. R is chosen to be 2^{1025}, which is greater than N and the greatest common divisor (GCD) of R and N is 1. Around 10,000 randomly chosen inputs were considered within the range 0 to $N-1$. Since there is a predominant effect of noise, we determine the average execution time for each input a from 1000 runs.

Listing 9.1 Pseudocode for modular squaring using Montgomery multiplication

```
1  int sqmult(int a, int R, int N)
2  {
3      A = (a * R) % N;
4
5      S = A * A;
6      S = (S + (((S * N^{-1}) % R) * N)) / R;
7      if (S > N)
8        S = S - N;
9      return S;
10
11 }
```

The conditional branch instruction occurs due to the *if* statement in line 8 of the listing. This causes two different execution paths:

- Branch *not-taken* path occurs when $S > N$. This causes the execution of the subtraction $S \leftarrow S - N$. In this case, there is no deviation from the sequential execution of the program.
- Branch *taken* path occurs when $S \leq N$. No subtraction is done and the execution jumps to the return statement following the "if" block. This is the branch target statement.

A part of the assembly code corresponding to Listing 9.1 is illustrated in Listing 9.2 for better understanding. The function call to the GMP function `__gmpz_cmp` makes the comparison between S and N. On its return, the `eax` register contains a positive, negative, or zero value corresponding to `A > N`, `A < N`, and `A = N`, respectively. The instruction `testl` compares this value with zero, while `jle` branches to the label `L4` if `eax` is less than or equal to zero, thereby not performing the subtraction. If `eax` is greater than zero, then the subtraction (using function `__gmpz_sub`) is performed.

Listing 9.2 Partial Assembly Code for the Squaring Function (sqmult)

```
    movq  %rdx, %rsi
    movq  %rax, %rdi
    call  __gmpz_cmp
    testl %eax, %eax
    jle .L4
    movl  $N, %edx
    movl  $S, %esi
    movl  $S, %edi
    call  __gmpz_sub
.L4:
    leave
    .cfi_def_cfa 7, 8
    ret
    .cfi_endproc
```

Since the inputs a were randomly chosen, some cause the branch to be taken while others do not. We classify the execution time into two sets depending on whether the branch was taken or not-taken. Figure 9.1 shows the average execution time for the function corresponding to inputs where the branch was not taken and Fig. 9.2 shows the average execution time where the branch was taken.

In both graphs, we see a separation in the execution time. The regions of the graphs with average execution time between 11,000 and 11,500 correspond to inputs where the branch prediction was correct and the regions above 12,000 correspond to the mispredictions. A clear separation is observed in both graphs. This experiment will be helpful for understanding the attack algorithms exploiting the branch mispredictions. The following sections discuss the details of these attacks.

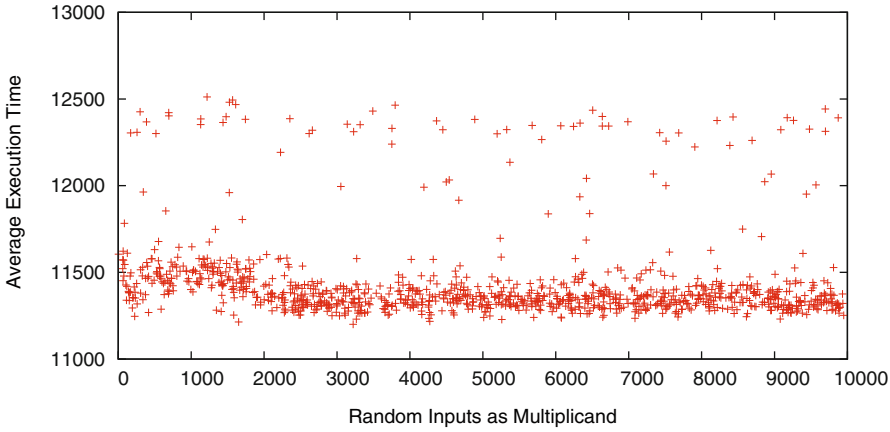


Fig. 9.1 Execution time from not-taken branches in Montgomery multiplication (on Intel i5). The *lower band* (around 11,500 clock cycles) indicates runs where the branches were correctly predicted not-taken. The *upper band* (around 12,500 clock cycles) indicates mispredictions

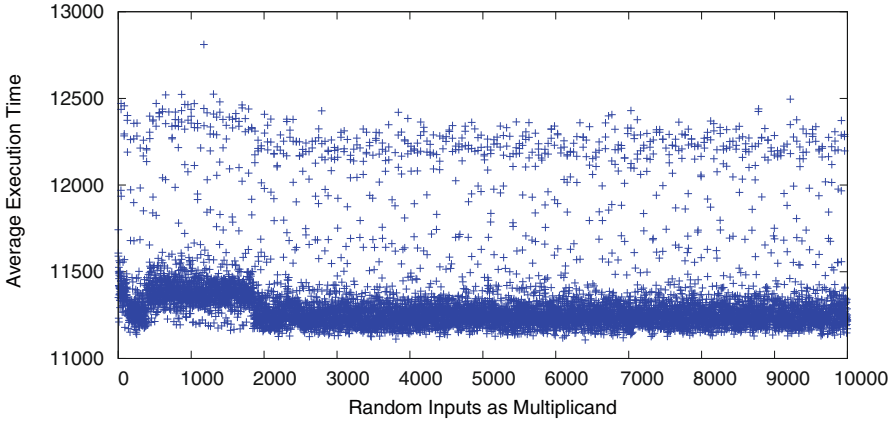


Fig. 9.2 Execution time from taken branches in Montgomery multiplication (on Intel i5). The *lower band* (around 11,000–11,500 clock cycles) indicates runs where the branches were correctly predicted taken. The *upper band* (around 12,500 clock cycles) indicates mispredictions

9.3 Attacking the Square and Multiply Exponentiation Algorithm

In this section, we show how timing mispredictions can be used to recover the secret exponent in the square and multiply algorithm (Algorithm 9.1) if the Montgomery multiplication algorithm is used. We assume that the adversary can invoke the function implementing the algorithm with different values of M and monitor the execution time. She however cannot determine the intermediate and any other internal values of the function.

The most significant bit of the exponent (i.e., d_0) in Algorithm 9.1 is set to one. The attacker determines all other bits iteratively. That is, bit d_i is determined only after d_{i-1} is found. The process to determine bit d_i is as follows:

Assumptions: The attacker knows the branch prediction algorithm used in the attack system. She also knows the initial state of the branch predictor.

Offline Phase: The attacker selects a large set of plaintexts $\mathcal{M}$ and then performs the following operations:

- Assumes that $d_i = 1$ and forms an exponent as follows: $d^{(1)} = d_0||d_1||d_2|| \cdots ||d_{i-1}||1$. The guess of $d_i = 1$ causes the multiplication in line 6 of Algorithm 9.1 to be performed in the ith iteration.
- Now, for each plaintext in $\mathcal{M}$, the attacker simulates Algorithm 9.1 to determine if there is a misprediction in the squaring in the $(i+1)^{th}$ iteration. Such an analysis is possible because the attacker knows how the branch predictor works.
- Based on whether there is a correct or wrong prediction in the Montgomery multiplier (MM), the set of plaintexts $\mathcal{M}$ can be partitioned into two sets: $\mathcal{M}_1$ and $\mathcal{M}_2$. These sets are defined as follows:
  - $\mathcal{M}_1 = \{m | m$ causes a misprediction during MM of $(i+1)^{th}$ squaring if $d_i = 1\}$
  - $\mathcal{M}_2 = \{m | m$ does not cause a misprediction during MM of $(i+1)^{th}$ squaring if $d_i = 1\}$
- The attacker now assumes $d_i = 0$ and forms an exponent as follows: $d^{(0)} = d_0||d_1||d_2|| \cdots ||d_{i-1}||0$. The above steps are repeated to partition the set of plaintexts $\mathcal{M}$ into two sets, $\mathcal{M}_3$ and $\mathcal{M}_4$. These sets are defined as follows:
  - $\mathcal{M}_3 = \{m | m$ causes a misprediction during MM of $(i+1)^{th}$ squaring if $d_i = 0\}$
  - $\mathcal{M}_4 = \{m | m$ does not cause a misprediction during MM of $(i+1)^{th}$ squaring if $d_i = 0\}$

Now equipped with four sets of plaintexts $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}_3$, and $\mathcal{M}_4$, the attacker is all set for the online phase.

Online Phase: Corresponding to each set $\mathcal{M}_i$ $(1 \leq i \leq 4)$, the attacker determines four average execution times for Algorithm 9.1. Since the ith bit of the exponent is either 0 or 1, two of the sets will show a separation in timing. For instance, consider the correct value of d_i is 1. This means,

- All plaintexts in $\mathcal{M}_1$ have a misprediction during MM of $(i+1)^{th}$ squaring. Thus, these plaintexts would correspond to a longer time.
- All plaintexts in $\mathcal{M}_2$ have no misprediction during MM of $(i+1)^{th}$ squaring. Thus, these plaintexts would correspond to a shorter time.
- In both $\mathcal{M}_3$ and $\mathcal{M}_4$, there are some plaintexts that would have a misprediction during MM of $(i+1)^{th}$ squaring, while other plaintexts in these sets will not have the misprediction.

When the average time is computed, there would be a separation between the time for $\mathcal{M}_1$ and $\mathcal{M}_2$ but no (or lesser) separation between $\mathcal{M}_3$ and $\mathcal{M}_4$. Algorithm 9.4 shows the details of the attack.

Algorithm 9.4: Determining bit d_i of the exponent

```
   Output: M, M_1, M_2, M_3, M_4
   Output: d_i
 1 begin
 2     C_j ← 0 for 1 ≤ j ≤ 4
 3     T_j ← 0 for 1 ≤ j ≤ 4
 4     for at-least 2^16 times do
 5         x ← choose random plaintext from M
 6         t_1 ← start time
 7         ScalarMultiply(x)
 8         t_2 ← end time
 9         for k ∈ {1,2,3,4} do
10             if x ∈ M_k then
11                 C_k ← C_k + 1
12                 T_k ← T_k + (t_2 − t_1)
13             end
14         end
15     end
16     A_j ← T_j/C_j for 1 ≤ j ≤ 4
17     if (A_1 − A_2) > (A_3 − A_4) then
18         return 1
19     end
20     else
21         return 0
22     end
23 end
```

In the algorithm, a random plaintext $\mathbf{x}$ is chosen from $\mathcal{M}$ and the scalar multiplication invoked and timed. The measured time is added to all sets where $\mathbf{x}$ is present. Since $\mathcal{M} = \mathcal{M}_1 \cup \mathcal{M}_2$ and $\mathcal{M} = \mathcal{M}_3 \cup \mathcal{M}_4$, $\mathbf{x}$ would be present in exactly two of the sets.

The average time $\mathcal{A}_i$ is found for each set and then the separation is determined. If there is more separation between $\mathcal{A}_1$ and $\mathcal{A}_2$, then d_i is 1, otherwise d_i is 0.

Evaluation: There are two limitations of this attack technique. First, the attacker needs to know the branch predictor algorithm present in the system. Second, she has to know the initial state of the branch predictor. Both these requirements are needed to partition $\mathcal{M}$ into sets $\mathcal{M}_1$, $\mathcal{M}_2$, $\mathcal{M}_3$, and $\mathcal{M}_4$.

Attackers may be able to tackle the initial state requirement of the predictor, for instance, by performing dummy operations that force the predictor to a particular state—such as branch taken. The first requirement however is more difficult to fulfill especially for commercially available processors. Branch prediction algorithms in modern commercial processors are a closely guarded secret and most manufactures do not reveal their internals. Thus, attackers would only be able to make smart guesses about the algorithm or use approximations.

In the experiments we consider an Intel i5 machine. We do not know the branch prediction internals in the i5 but consider the 2-bit bimodal algorithm (see

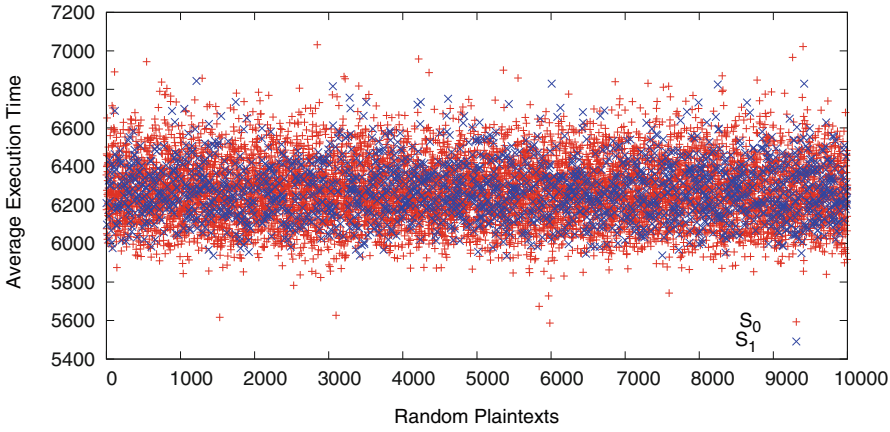


Fig. 9.3 Timing variation due to plaintexts for sets $\mathcal{M}_3$ and $\mathcal{M}_4$ when $d_i = 1$—the wrong guess

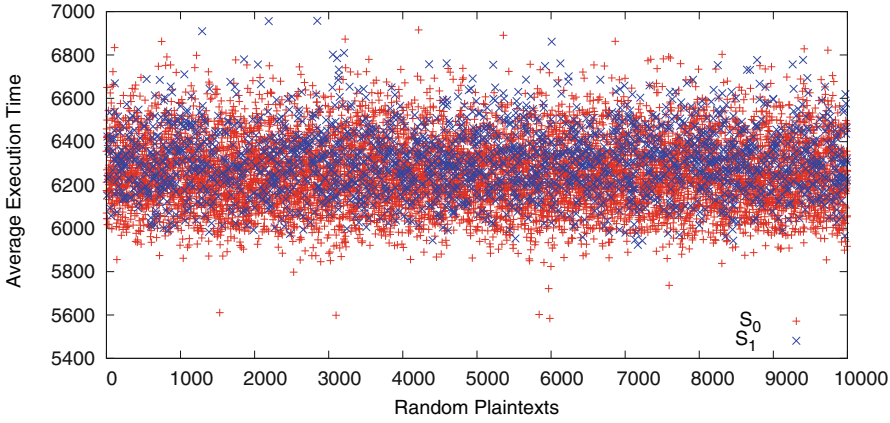


Fig. 9.4 Taming variation for random plaintexts for sets $\mathcal{M}_1$ and $\mathcal{M}_2$ when $d_i = 1$—the correct guess

Sect. 3.3.2.2), which we assume is a close match. Figure 9.3 shows the average execution time for the exponentiation (Algorithm 9.1) for plaintexts $\mathcal{M}_3$ and $\mathcal{M}_4$ and Fig. 9.4 shows the average execution time for plaintexts in $\mathcal{M}_1$ and $\mathcal{M}_2$.

Table 9.1 tabulates the difference in the average separation between the sets. It can be observed that the difference in timing for the wrong assumption is small whereas the difference in timing for the plaintext partitions for the correct guess of d_i is significantly large. Thus, in spite of the assumed predictor, the attack was able to correctly identify d_i as 1. We expect that as better models for the predictor emerge, this attack's success would improve considerably. Alternatively, the attacker can use other attack strategies. Some of these strategies are discussed in the following sections.

Table 9.1 Separation in timing for assumptions $d_i = 0$ or $d_i = 1$

Assumption	Separation (in clock cycles)	
$d_i = 0$	$(\mathcal{A}_3 - \mathcal{A}_4) = 6278 - 6271 = 7$	*Wrong guess* Average separation negligible
$d_i = 1$	$(\mathcal{A}_1 - \mathcal{A}_2) = 6312 - 6257 = 55$	*Correct guess* Average separation significant

9.4 Asynchronous Attack on the Square and Multiply Algorithm

In a simultaneous multithreading (SMT) environment, a single processor core is simultaneously shared between two processes. The processor does not distinguish between processes from different users. This could lead to attacks on the RSA exponentiation using the square and multiply algorithm (Algorithm 9.1).

The asynchronous attack we describe now does not require knowledge of the branch predictor algorithm or its initial state. It uses the fact that when the Montgomery multiplication (Sect. 9.1.3) executes, information about the branch taken or not-taken is stored in the branch target buffer (BTB). The BTB is a cache shared between all processes in the system. It is of limited size, therefore a branch instruction in a process may evict a previous entry in the BTB. The attack works as follows:

The adversary runs a spy process simultaneously with the exponentiation. The spy process uses dummy branch instructions to continuously evict the contents of the BTB. This induces mispredictions in the square and multiply process, which is manifested in its execution time. As in the previous attack, the attack is iterative. Bit d_i in the exponent is determined only after bits d_0–d_{i-1} are known. Below we describe how the bit d_i is obtained.

Offline Phase: Similar to the offline phase in Sect. 9.3, the attacker selects a large set of plaintexts $\mathcal{M}$ and creates four sets from it as follows:

- $\mathcal{M}_1 = \{m | m \in \mathcal{M}$ causes the branch to be taken during MM of $(i+1)^{th}$ squaring if $d_i = 1\}$
- $\mathcal{M}_2 = \{m | m \in \mathcal{M}$ causes the branch not to be taken during MM of $(i+1)^{th}$ squaring if $d_i = 1\}$
- $\mathcal{M}_3 = \{m | m \in \mathcal{M}$ causes the branch to be taken during MM of $(i+1)^{th}$ squaring if $d_i = 0\}$
- $\mathcal{M}_4 = \{m | m \in \mathcal{M}$ causes the branch not to be taken during MM of $(i+1)^{th}$ squaring if $d_i = 0\}$

As in the attack in Sect. 9.3, $\mathcal{M}_1$ and $\mathcal{M}_2$ partition $\mathcal{M}$. Similarly, $\mathcal{M}_3$ and $\mathcal{M}_4$ partition $\mathcal{M}$. However, there is a subtle difference between the two offline phases. The attack in Sect. 9.3 required the partitions to be based on correct and wrong branch predictions. However, in the asynchronous attack described here, the partitions are based on whether the branch is taken or not taken in the $(i+1)^{th}$ iteration.

Online Phase: In the online phase, the attacker executes a spy process simultaneously with the cipher in a multithreaded environment and times the execution of the exponentiation. The spy is capable of evicting the contents of the BTB. Since the BTB stores information about the branch targets of the Montgomery multiplication used during the exponentiation, evicting the BTB would evict these branch target entries.

The aim of the spy is to evict the BTB entries just before the $(i+1)^{th}$ iteration. In the $(i+1)^{th}$ iteration, when the CPU executes the conditional reduction branch instruction in the Montgomery squaring, it is unable to find the branch target address in the BTB. This may result in a compulsory misprediction if the branch is taken.

Depending on which set the plaintext belongs to and the value of d_i, the following will happen. Let us say, $d_i = 1$, then

- All plaintexts in $\mathcal{M}_1$ take the branch in the $(i+1)^{th}$ squaring. Therefore, these plaintexts would cause a compulsory misprediction. Thus the execution time is longer.
- All plaintexts in $\mathcal{M}_2$ do not take the branch in the $(i+1)^{th}$ squaring. Therefore, these plaintexts would not have a misprediction (since we assume the default is branch not taken). Thus, the execution time is shorter.
- In both $\mathcal{M}_3$ and $\mathcal{M}_4$, there are some plaintexts that take the branch while others do not. Thus, the forced misprediction does not occur all the time.

When the average time is computed, there would be a separation between the time for $\mathcal{M}_1$ and $\mathcal{M}_2$ but no (or lesser) separation between $\mathcal{M}_3$ and $\mathcal{M}_4$.

If $d_i = 0$, the separation in the average execution time is expected to occur between $\mathcal{M}_3$ and $\mathcal{M}_4$ instead of $\mathcal{M}_1$ and $\mathcal{M}_2$. The attack process is exactly similar to Algorithm 9.4.

Evaluation: The critical factor in the entire attack is the spy process that evicts the BTB. The spy needs to flush the BTB just before the squaring in the $(i+1)^{th}$ iteration. Since the exact instant when the $(i+1)^{th}$ iteration is executed is not known, the spy has to periodically flush the BTB and hope that a flush happens before the target iteration.

The resolution of clearing a BTB has a major impact in determining the attack's success. Either the BTB can be evicted completely or partially. Each has its own advantages.

- *Total eviction.* In this method, the entire BTB is cleared by the adversary. This is the easiest eviction strategy because the knowledge of the target destination address in the Montgomery multiplication is not required. However in practice, clearing the entire BTB between two consecutive squaring operations is unrealistic since it requires more time than performing the operations between two consecutive squarings. To improve the performance of the attack, the adversary can evict the BTB partially.
- *Partial eviction.* The adversary continuously clears only a part of the BTB that stores the BTB set of the target address of the branch taken. In the extreme case, the adversary only clears the particular set of the BTB that stores the target address

of the target branch. In these cases, the adversary needs to know how the branch targets get mapped in the BTB.

In this attack, it is assumed that the adversary does not have any means to control a particular misprediction event since the synchronization between the spy and the attacker was deliberately avoided. In the next section, a synchronized version of the explained attack is presented.

9.5 Synchronous Attack on the Square and Multiply Algorithm

The main idea of the synchronous attack that reveals d_i are the following observations:

1. In the ith iteration of Algorithm 9.1, if $d_i = 1$, then the branch is not taken and the multiplication in line 5 is done. If $d_i = 0$, the branch is taken and the multiplication is skipped.
2. If the processor is forced to predict branch not taken, $d_i = 0$ will cause a misprediction, thus resulting in a pipeline flush and an increase in execution time. On the other hand, $d_i = 1$ will not cause any misprediction, therefore no increase in execution time.
3. The adversary forces the branch not taken state in the processor by evicting the branch target in the BTB just before the ith iteration.

The attack works as follows:

- The attacker chooses an M and determines the execution time for Algorithm 9.1, say T_1.
- The execution is run again with the same M and the execution time is measured. Let T_2 be the time. The only difference compared to the first execution is that just before the ith iteration, the adversary clears a single target location of the BTB corresponding to the branch instruction.
- if $(|T_2 - T_1|)$ is greater than some threshold τ, then $d_i = 0$ else $d_i = 1$.

In all the attacks discussed so far, the attacker measures the execution time of the misprediction while running an implementation of the square and multiply algorithm. In the following section, a new attack strategy is explored which is asynchronous in nature and measures the execution time of the branches of a spy process.

9.6 Trace Driven Attack Targeting the BTB

This attack is similar to that of the prime and probe attack on cache memories (Sect. 8.1). Just like the prime and probe attack, the adversary executes a spy program. The spy has a series of branch instructions in a loop, which are executed and timed. The execution time for the branch instructions would initially be large due to several mispredictions that occur. The execution time would reduce gradually as the BTB gets filled. When a cipher executes simultaneously with the spy, some of the BTB entries of the spy get evicted. This forces branch mispredictions in the spy causing

an increase in the execution time. The increase in time leaks information about the cipher's secret key. The steps in the attack are enumerated below:

- The adversary initiates the attack by allowing the spy process to execute a number of branch statements that gradually fill the BTB with branch target addresses from the spy process.
- After a while, the cipher is allowed to execute.
- When the exponentiation code in the cipher encounters a branch, the CPU will not find the target address of the branch in the BTB. The prediction then defaults to *not-taken*.
- If $d_i = 0$, the conditional branch in the square and multiply algorithm (Algorithm 9.1) results to *taken* and one of the entries in the BTB corresponding to the spy process is evicted in order to accommodate the new target destination. If $d_i = 1$, then there is no misprediction, thus no entries in the BTB get evicted.
- After the cipher completes execution, the spy reexecutes its branches and measures the time required. If the branches take a longer time than usual, it is attributed to the mispredictions in the BTB due to the cipher's evictions. Thus, d_i is taken 0. If there is no change in the execution time of the branches, it is attributed to no mispredictions and d_i is taken 1.

9.7 Conclusion

Branches in programs can lead to information leakage due to the branch prediction units present in modern processors. Eliminating branches from programs is not always possible, thus alternate approaches to counter branch prediction based timing attacks need to be determined. The next chapter provides an overview of the countermeasures available for timing attacks. Many of these countermeasures can be applied or tuned to prevent branch prediction attacks.

References

1. Aciiçmez O, Koç ÇK, Seifert JP (2006) On the power of simple branch prediction analysis. IACR Cryptology ePrint Archive, vol 2006, p 351
2. Aciiçmez O, Gueron S, Seifert JP (2007) New branch prediction vulnerabilities in OpenSSL and necessary software countermeasures. In: Galbraith SD (ed) IMA international conference on security lecture notes in computer science, vol 4887. Springer, pp 185–203
3. Aciiçmez O, Koç ÇK, Seifert JP (2007) Predicting secret keys via branch prediction. In: Abe M (ed) CT-RSA, ser. lecture notes in computer science, vol 4377. Springer, pp 225–242

Chapter 10
Countermeasures for Timing Attacks

The criteria for a timing attack to be successful are the following:

1. The run time behavior of the crypto application should vary depending on the secret key. For instance, the instructions executed or memory locations accessed should depend on the secret. Even the data operated on could also cause run time behavior to vary.
2. These variations in the run time behavior must be manifested through the time required to execute operations.
3. The adversary must have sufficient capabilities to monitor time variations. For instance, the adversary should have access to the system executing the crypto application.
4. A clock source precise enough to capture variations in the run time behavior of the application should be available.
5. A synchronization signal to determine when the application starts and/or completes its processing.
6. In addition to these criteria, the success obtained in the attack largely depends on the number of time measurements made.

An attack would be successful only if all these criteria are met. It is therefore not surprising that countermeasures try to make one or more of these criterias difficult to fulfill. Completely preventing a criterion can be done, for example, by eliminating the cache memory from processors. However, this leads to significant performance degradation, which far supersedes the security gains obtained. Thus, the aim of practical countermeasures is to obtain the right balance between security and performance. The upcoming sections discuss the countermeasures.

10.1 Application Level Countermeasures

Countermeasures in the crypto application are generally easy to apply and provide sufficient levels of security against timing attacks. However, they suffer from significant shortcomings. First, most of the application-level countermeasures are heavy

C. Rebeiro et al., *Timing Channels in Cryptography,*
DOI 10.1007/978-3-319-12370-7_10

and inefficient. Further, all applications require to be patched with the same countermeasures. A system may thus have several applications which are modified with these bulky countermeasures. This adversely affects performance and energy requirements of the system. Neverthe-less, these countermeasures provide a quick fix to prevent timing attacks. We therefore discuss some of these countermeasures here.

10.1.1 Countermeasures Involving Look-Up Tables

In block cipher implementations, key dependent look-up table operations are performed. While these operations improve speed, it leads to vulnerabilities through timing channels. One method to prevent the attack is designing new ciphers which have no s-boxes, for example [1]. For the general class of ciphers that use s-boxes, implementations without look-up tables would have a drastic affect on performance. An alternate implementation without look-up tables is using bitslicing [2, 3]. This allows multiple encryptions to be done concurrently in a single processor. Bitslicing yields high speed encryption, which however is restricted to certain nonfeedback modes of operation, and therefore cannot be universally applied.

An alternate approach is to implement the cipher with appropriately chosen look-up tables or restrict their usage. This approach tries to achieve security with increased performance by either restricting the size of the look-up tables or the accesses made to it.

10.1.1.1 Look-Up Tables in Selected Rounds

Cache attacks on block ciphers discovered so far, rely on collisions in the first few or the last round of the cipher. For instance in Advanced Encryption Standard (AES), cache attacks target collisions in the look-up tables either in the first, second, or the last round. It is only in these rounds that collisions can be exploited by an adversary.

Preventing usage of look-up tables in these "sensitive" rounds would therefore prevent the attack. For instance, a hybrid implementation can be built for AES, which uses look-up tables in all but the first, second, and final round. In these sensitive rounds alternate implementations can be used for the s-box, for example, by logical equations or by using small look-up tables as discussed in the next section. The execution time for these implementations of s-boxes must be independent of the secret key. This approach provides security because there are no timing variations in run-time execution due to the sensitive rounds. For instance, compared to an AES implementation with look-up tables, the overhead of an implementation with two rounds protected is 10.8 compared to 66.7 for an implementation that does not use look-up tables. The rounds implemented without look-up tables shield the implementation against cache attacks, while the look-up tables used in the intermediate rounds boost performance.

10.1.1.2 Small Look-Up Tables

Consider an implementation of a cipher with a look-up table of size $l \cdot 2^{\delta}$, where l is the number of memory blocks occupied by the table and 2^{δ} the number of elements in the table that share the same memory block. The number of bits required to access an element in the table is $\delta + \log_2 l$; $\log_2 l$ bits to select the memory block and δ bits to address the element within the block. Most cache attacks are not able to distinguish between elements that lie in the same block. Assuming that the table is aligned to a memory block, these cache attacks can only retrieve $\log_2 l$ bits leaving the remaining δ bits uncertain. The uncertainty can be increased by packing more elements into a memory block thereby reducing l.

In the extreme, if the entire look-up table fits in a single cache line, then the adversary cannot ascertain any information from the memory access patterns. In such cases $\log_2 l = 0$, that is the adversary retrieves 0 bits. The problem however is that, s-boxes used in most ciphers are considerably larger than the memory block. Therefore compressing them into a single block is not easy. In [4] and [5], techniques that compress s-boxes of standard ciphers like CLEFIA into a single memory block were proposed. One technique that is especially suited for s-boxes based on the multiplicative inverse in a finite field uses composite field isomorphisms [6]. This allows the use of smaller tables which stores subfield operations. For instance the AES s-box, which typically requires atleast 256 bytes, can be compressed into a single memory block of 32 or 4 bytes.

10.1.1.3 Cache Warming

Loading the contents of the look-up tables prior to encryption could reduce the risk of cache attacks. As a result, the accesses to the look-up table during the cipher execution would be cache hits. However, this alone is not sufficient to thwart attacks. Conflict misses that occur during the execution may evict the look-up table entries from the cache, potentially resulting in cache misses that leak information about the secret key. The system should therefore ensure that elements in the look-up table are not evicted during the encryption. This is not easy to satisfy.

10.1.1.4 Choosing Look-Up Tables

The number and size of the look-up tables play an important role in an attack's success. Choosing appropriate number and size of look-up tables can therefore be used to mitigate time-driven cache attacks. For a specific system, Chap. 6 shows how appropriate look-up tables can be chosen without compromising on the performance.

10.1.2 Data-Oblivious Memory Access Pattern

This allows the pattern of memory accesses to be performed in an order that is oblivious to the data passing through the algorithm. To a certain extent, modern microprocessors provide such data oblivious memory accesses by reordering instructions. For example, four memory accesses to different locations say, A, B, C, and D, would get executed in any of the four ways (say $BCDA$ or $DCAB$). This reordering would increase the difficulty of certain time-driven and access-driven attacks. However, the reordering is restricted to memory accesses which do not have data dependencies. For block ciphers, such independent memory accesses are present within a round but not across rounds, therefore only partially fulfills data-oblivious requirements.

A naïve method to attain complete data-oblivious memory accesses is to read elements from every memory block of the table, in a fixed order, and use just the one needed. Another option is to add noise to the memory access pattern by adding spurious accesses, for example, by performing a dummy encryption in parallel with the real one. A third option is to mask the sensitive table accesses with random masks that are stripped away at the end of the encryption. Alternatively, the table can be permuted regularly to thwart attacks using statistical analysis.

Generic program transformations are also present for hiding memory accesses [7]. However, there are huge overheads in performance and memory requirements. More practical proposals have been developed using shuffling [8] and permutations [9].

10.1.3 Constant and Random Time Implementations

A combination of the countermeasures discussed so far can be used to deliver constant time implementations [10, 11]. To obtain such implementations requires cache warming to ensure that all memory accesses result in cache hits. A timing probe which measures the current execution time for the cipher, and a compensation loop which is designed to increase the encryption time until the worst case execution time is obtained. Additionally, to prevent evictions of the table occurring during an interrupt, an interrupt detector is required to rewarm the cache.

As opposed to constant time implementations, adding a random delay to each execution time can be used to increase the attack difficulty. The countermeasure was first proposed for DPA in [12]. Tunstall and Benoit in [13] and later Coron and Kizhvatov in [14, 15] provide improvements by modifying the distribution of the delay.

10.2 Countermeasures Applied in the Hardware

Subtle changes in the hardware can make attacks much more difficult with little overhead in the performance. In this section we survey some of the hardware modifications that have been suggested to counter cache attacks.

10.2.1 Noncached Memory Accesses

Certain memory pages can be flagged as noncacheable. This would prevent memory accesses from loading data into the cache. Consequently, every memory access would read data from the RAM and every access would thereby be a cache miss preventing all cache attacks.

10.2.2 Specialized Cache Designs

Specialized cache memory designs have been proposed for thwarting cache attacks. They work on the fact that information leakage is due to sharing of cache resources, thus leading to *cache interference*. These solutions provide means of preventing access-driven attacks. Their effectiveness in blocking time-driven attacks has not yet been analyzed. In [16], Percival suggests eliminating cache interference by modifying the cache eviction algorithms. The modified eviction algorithms would minimize the extent to which one thread can evict data from another thread.

In [17], Page proposed to partition cache memory, which is a direct-mapped *partitioned cache* that can dynamically be partitioned into protected regions by the use of specialized cache management instructions. By modifying the instruction set architecture and tagging memory accesses with partition identifiers, each memory access is hashed into a dedicated partition. Such cache management instructions are however only available in the operating system. While this technique prevents cache interference from multiple processes, the cache memory is underutilized due to rigid partitions. For example, a process may use very few cache lines of its partition, but the unused cache lines are not available to another process.

In [18], Wang and Lee provide an improvement on the work by Page using a construct called *partition-locked cache* (PLCache), where the cache lines of interest are locked in cache, thereby creating a private partition. These locked cache lines cannot be evicted by other cache accesses not belonging to the private partition. In the hardware, each cache line requires additional tags comprising a flag to indicate if the line is locked and an identifier to indicate the owner of the cache line. The underutilization of Page's partitioned cache still persists because the locked lines cannot be used by other processes, even after the owner no longer requires them.

Wang and Lee also propose a *random-permutation cache* (RPCache) in [18], where, as the name suggests, it randomizes the cache interference, so that the difficulty of the attack increases. The design is based on the fact that information is leaked only when cache interference is present between two different processes. The architecture requires an additional hardware called the *permutation table*, which maps the set bits in the effective address to obtain new set bits. These are then used to index the cache set array. Changing the contents of the permutation table will invalidate the respective lines in the cache. This causes additional cache misses and a randomization in the cache interference. In [19], Wang and Lee use an underlying direct-mapped cache and dynamically reprogrammable cache mapping algorithms to achieve randomization. From the security perspective, this technique is shown to be as effective in preventing attacks as RPCache, but with less overheads on the performance. In [20], Kong et al. show that partition locked and random permutation caches, although effective in reducing performance overhead, are still vulnerable to advanced cache attack techniques. They then go on to propose modifications to Wang and Lee's proposals to improve the security [21, 22].

In [23], Domnitser et al. provide a low cost solution to prevent access-driven attacks based on the fact that the cipher evicts one or more lines of the spy data from the cache. The solution, which requires small modifications of the replacement policies in cache memories, restricts an application from holding more than a predetermined number of lines in each set of a set-associative cache. With such a cache memory, the spy can never hold all cache lines in the set, therefore the probability that the cipher evicts spy data is reduced. By controlling the number of lines that the spy can hold, trade-off between performance and security can be achieved.

10.2.3 Specialized Instructions

A few recent microprocessors support specialized instructions for ciphers. For example, Intel supports the AES-NI for their new processors [24], which allows AES encryption or decryption using a combination of six instructions. These instructions have dedicated hardware support [25] and therefore do not require to use look-up tables. Dedicated hardware has the added advantage of boosting performance. With Intel's AES-NI, speed of encryptions can be increased more than an order of magnitude. The drawback however is that, such dedicated hardware are only applicable for AES-based crypto systems. There are several other ciphers that are used in many applications, which cannot gain from these instructions. These ciphers are still vulnerable to attacks. An alternate direction is to develop improved instruction sets, which would be friendly for general cipher algorithms. For example, bit-permutation instructions were suggested in [26, 27] as a means to speedup a typical block cipher. For public-key ciphers, several instruction enhancements have been suggested such as [28–30]. In [31], a new instruction to be added to the microprocessor ISA called PERMS, was suggested. This instruction allows any arbitrary permutation of the bits

in an n-bit word, and can be used to accelerate diffusion layer implementations in block ciphers, thereby providing high-speed encryptions.

10.2.4 Hardware Prefetching

Prefetching data automatically into cache memory is present in modern microprocessors to reduce the memory latencies by anticipating accesses. In [32], automatic hardware prefetching is suggested as a means to counter cache attacks, since it would confuse the adversary while distinguishing between a cache hit and a miss.

While prefetching was in fact demonstrated to make access-driven attacks slightly more difficult [33]. It has the opposite effect in profiled time-driven cache attacks. In profiled time-driven attacks such as the one by Bernstein in 2005, prefetching can be a source of information leakage rather than a means to mitigate the attack.

10.2.5 Fuzzying Clocks

Current timing attacks require distinguishing between events by their execution time. This requires highly accurate timing measurements to be made. A popular method for making these measurements is by the rdtsc (*read time stamp counter*) instruction, which is present in most modern day processors. This instruction allows the processor's time stamp counter (TSC) to be read.

Since the TSC is incremented in every clock cycle, rdtsc can provide nanoscale accuracy. Denying the adversary access to the rdtsc instructions robs her of a simple and highly precise clock source. However, the main drawback preventing user access to the TSC is that several benign applications (such as Linux kernels, multimedia games, and certain cryptographic algorithms [34]) rely on rdtsc. These applications will cease to function if the instruction is disabled. A less stringent way is to therefore *fuzz* the timestamp returned by the rdtsc instruction. This is possible because unlike timing attacks, benign applications do not require highly accurate timing measurements. Time fuzzing can be either done by reducing the accuracy of the measurement (such as by masking least significant bits of the timestamp [35]) or by reducing the precision of the measurement (such as by injecting noise into the timestamp [36]).

If $\mathtt{t1}$ is the current value of the TSC, then masking returns $\lfloor \mathtt{t1}/\mathtt{E} \rfloor$ ($\mathtt{E}$ is a time duration called *epoch*). Consequently, $\mathtt{t}$ has the form $\lfloor \frac{\mathtt{t2}}{\mathtt{E}} \rfloor - \lfloor \frac{\mathtt{t1}}{\mathtt{E}} \rfloor$. The size of the epoch is crucial. It should be large enough to protect against timing attacks, yet small enough to not affect benign applications. In [37], Vattikonda et al. propose a way to bypass such masking schemes. The technique uses the fact that timing attacks do not need absolute timing measurements, rather it is only required to distinguish between two or more events. In the proposal, the adversary first loops continuously to synchronize with the start of the epoch. This is denoted by $\mathtt{t1e} = \lfloor \frac{\mathtt{t1}}{\mathtt{E}} \rfloor$. Then the operation to be timed is called and an rdtsc instruction is invoked in a tight loop till

the end of the epoch (detected by the change in the rdtsc value). A counter c counts the number of rdtsc invocations made. The value of c in the end of the loop can be used to distinguish between events executed by the operation, as a smaller value of $t2e = \lfloor \frac{t2}{E} \rfloor$ will lead to a larger value of c.

```
c = 0;
/* Synchronize to the start of epoch */
t1e = rdtsc;
while(t1e != rdtsc);
function() /* the operation that needs to be timed */
/* Continuously scan TSC until end of epoch */
t2e = rdtsc;
while(t2e != rdtsc) c = c + 1;
return c and (t2e - t1e)
```

In the noise injection technique, a random offset between $[0, E)$ is added to the timestamp value returned by rdtsc. There are however two possible problems. The first is likely to occur when multiple rdtsc instructions are invoked. The first invocation of rdtsc returns a value of the form $t1 + r1$, while the second invocation return has the form $t2 + r2$, where $t1$ and $t2$ are the timestamp values and $r1$ and $r2$ are the random numbers ($0 \leq r_1, r_2 < E$). If the two invocations are done in quick succession then $t1 = t2$, while it is possible that $r1 > r2$. This gives the impression of time moving backward. The second drawback of the noise injection scheme is that the randomness is limited by the size of E. If sufficient number time measurements are made and average taken, the effect of the injected noise can be eliminated.

In [34], Martin, Demme, and Sethumadhavan propose a scheme called *Timewarp*. The scheme prevents the attack in [37] by restricting the TSC reading to the end of an epoch. Additionally, the rdtsc instruction is delayed by a random time (less than an epoch) before the result is returned. Noise is further added to the TSC value to increase the difficulty of the attack.

Even without using the rdtsc instructions, it is possible to make highly accurate timing measurements. For example, on multiprocessor systems, a counter can be run in a tight loop in a different thread, which can be used to make sufficiently precise timing measurements. These are called *virtual time-stamp counters* (VTSC). To prevent use of such counters, [34] proposes to add hardware in the processor, which detects these loops and injects random delays into them.

10.3 Countermeasures in the Operating System

While hardware countermeasures are expensive and application layer countermeasures are heavy, the search is still on for the countermeasure that is a panacea. The operating system may hold the key—especially in attacks that use a spy process to snoop at a victim's execution behavior.

In the operating system for instance, the scheduler can be tweaked to ensure processes from different security domains do not share the same hardware resources. Fine-grained attacks, which exploit scheduler policies may be mitigated by changes to the scheduling algorithm. For instance, by ensuring that a minimum threshold is forced before a context switch happens between two processes.

10.4 Conclusion

Countering timing attacks can be done at either the application layer, hardware, or in the operating system. In each category there are several options available. While application layer countermeasures can easily be applied, they have huge overheads that affect system performance. On the other hand, countermeasures applied at the hardware level are efficient but are expensive requiring system redesign. Comparatively, countermeasures in operating systems and virtual managers are less explored.

Engineers would need to make learned decisions on what countermeasures to incorporate. These decisions should be made early in the design phase. Trade-offs between the performance, cost, and security need to be considered. However, this is easier said than done. A countermeasure which works well with for one system and algorithm may not be suitable in another environment. While performance and costs can be easily gauged, the same cannot be done with security. For this reason, metrics need to be developed that could quantify the security of the system. For time-driven cache attacks, this book provided a means to do this. Similar frameworks need to be developed for other leakage models and attacks. The hope is for a unified framework that could assess the vulnerability of a system to all forms of timing attacks.

References

1. Klimov A, Shamir A (2002) A new class of invertible mappings. In: Çetin Kaya Koç Jr BSK Paar C (eds) CHES. Lecture notes in computer science, vol 2523. Springer, Berlin, pp 470–483
2. Biham E (1997) A fast new DES implementation in software. In: Biham E (ed) FSE. Lecture notes in computer science, vol 1267. Springer, Berlin, pp 260–272
3. Rebeiro C, Selvakumar AD, Devi ASL (2006) Bitslice implementation of AES. In: Pointcheval D, Mu Y, Chen K (eds) CANS. Lecture notes in computer science, vol 4301. Springer, Berlin, pp 203–212
4. Rebeiro C, Mukhopadhyay D (2011) Cryptanalysis of CLEFIA using differential methods with Cache Trace Patterns. In: Kiayias A (ed) CT-RSA. Lecture notes in computer Science, vol 6558. Springer, Berlin, pp 89–103
5. Rebeiro C, Poddar R, Datta A, Mukhopadhyay D (2011) An enhanced differential cache attack on CLEFIA for large cache lines. In: Bernstein DJ, Chatterjee S (eds) INDOCRYPT. Lecture notes in computer science, vol 7107. Springer, Berlin, pp 58–75
6. Paar C (1994) Efficient VLSI architectures for bit-parallel computation in Galois fields. Ph.D. dissertation, Institute for Experimental Mathematics, Universität Essen, Germany, June 1994
7. Goldreich O, Ostrovsky R (1996) Software protection and simulation on oblivious RAMs. J ACM 43(3):431–473

8. Zhuang X, Zhang T, Lee H-HS, Pande S (2004a) Hardware assisted control flow obfuscation for embedded processors. In: Irwin MJ, Zhao W, Lavagno L, Mahlke SA (eds) CASES. ACM, New York, pp 292–302
9. Zhuang X, Zhang T, Pande S (2004b) HIDE: an Infrastructure for efficiently protecting information leakage on the address bus. In: Mukherjee S, McKinley KS (eds) ASPLOS. ACM, New York, pp 72–84
10. Bernstein DJ (2005) Cache-timing attacks on AES. Technical report, 2005
11. Canteaut A, Lauradoux C, Seznec A (2006) Understanding cache attacks. INRIA, Research Report RR-5881, 2006. http://hal.inria.fr/inria-00071387/en/
12. Clavier C, Coron J-S, Dabbous N (2000) Differential power analysis in the presence of hardware countermeasures. In: Çetin Kaya Koç Paar C (eds) CHES. Lecture notes in computer science, vol 1965. Springer, Berlin, pp 252–263
13. Tunstall M, Benoît O (2007) Efficient use of random delays in embedded software. In: Sauveron D, Markantonakis C, Bilas A, Quisquater J-J (eds) WISTP. Lecture notes in computer science, vol 4462. Springer, Berlin, pp 27–38
14. Coron J-S, Kizhvatov I (2009) An efficient method for random delay generation in embedded software. In: Clavier C, Gaj K (eds) CHES. Lecture notes in computer science, vol 5747. Springer, Berlin, pp 156–170
15. Coron J-S, Kizhvatov I (2010) Analysis and improvement of the random delay countermeasure of CHES 2009. In: Mangard S, Standaert F-X (eds) CHES. Lecture notes in computer science, vol 6225. Springer, Berlin, pp 95–109
16. Percival C (2005) Cache missing for fun and profit. In: Proceedings of BSDCan 2005, Ottawa
17. Page D (2005) Partitioned cache architecture as a side-channel defence mechanism. IACR cryptology eprint archive, vol 2005, p 280
18. Wang Z, Lee RB (2007) New cache designs for thwarting software cache-based side channel attacks. In: Tullsen DM, Calder B (eds) ISCA. ACM, New York, pp 494–505
19. Wang Z, Lee RB (2008) A novel cache architecture with enhanced performance and security. In: MICRO. IEEE Computer Society, pp 83–93
20. Kong J, Aciiçmez O, Seifert J-P, Zhou H (2008) Deconstructing new cache designs for thwarting software cache-based side channel attacks. In: Jaeger T (ed) CSAW. ACM, New York, pp 25–34
21. Kong J, Aciicmez O, Seifert J-P, Zhou H (2012) Architecting against software cache-based side channel attacks. IEEE Transactions on Computers, vol 99, no. PrePrints
22. Kong J, Aciiçmez O, Seifert J-P, Zhou H (2009) Hardware-software integrated approaches to defend against software cache-based side channel attacks. In: HPCA. IEEE Computer Society, pp 393–404
23. Domnitser L, Jaleel A, Loew J, Abu-Ghazaleh NB, Ponomarev D (2012) Non-monopolizable caches: low-complexity mitigation of cache side-channel attacks. TACO 8(4):35
24. Gueron S (2010) Intel Advanced Encryption Standard (AES) instructions set (Rev : 3.0)
25. Mathew S, Sheikh F, Agarwal A, Kounavis M, Hsu S, Kaul H, Anders M, Krishnamurthy R (2010) 53Gbps native $GF(2^4)^2$ composite-field AES-encrypt/decrypt accelerator for content-protection in 45nm high-performance microprocessors. In: VLSI Circuits (VLSIC), 2010 IEEE Symposium on. 16–18 June 2010, pp 169–170
26. Shi ZJ, Yang X, Lee RB (2008) Alternative application-specific processor architectures for fast arbitrary bit permutations. IJES 3(4):219–228
27. Lee RB, Shi Z, Yin YL, Rivest RL, Robshaw MJB (2004) On permutation operations in cipher design. In: ITCC (2). IEEE Computer Society, pp 569–577
28. Großschadl J, Kamendje G-A (2003) Instruction set extension for fast elliptic curve cryptography over binary finite fields $GF(2^m)$. In: ASAP. IEEE Computer Society, pp 455–468
29. Großschadl J, Savas E (2004) Instruction set extensions for fast arithmetic in finite fields $GF(p)$ and $GF(2^m)$. In: Joye M, Quisquater J-J (eds) CHES. Lecture notes in computer science, vol 3156. Springer, Berlin, pp 133–147
30. Vejda T, Page D, Grossschadl J (2007) Instruction set extensions for pairing-based cryptography. In: Takagi T, Okamoto T, Okamoto E, Okamoto T (eds) Pairing. Lecture notes in computer science, vol 4575. Springer, Berlin, pp 208–224

31. Khurana S, Kolay S, Rebeiro C, Mukhopadhyay D (2013) Light weight cipher implementations on embedded processors. In: Design and technology of integrated systems (DTIS). IEEE Computer Society
32. Lauradoux C (2005) Collision attacks on processors with cache and countermeasures. In: Wolf C, Lucks S, Yau P-W (eds) WEWoRC. LNI, vol 74. GI, Bonn, pp 76–85
33. Demme J, Martin R, Waksman A, Sethumadhavan S (2012) Side-channel vulnerability factor: a metric for measuring information leakage. In: ISCA. IEEE, pp 106–117
34. Martin R, Demme J, Sethumadhavan S (2012) TimeWarp: rethinking timekeeping and performance monitoring mechanisms to mitigate side-channel attacks. In: ISCA. IEEE, pp 118–129
35. Osvik DA, Shamir A, Tromer E (2005) Cache attacks and Countermeasures: the case of AES. Cryptology ePrint Archive, Report 2005/271
36. Jayasinghe D, Fernando J, Herath R, Ragel R (2010) Remote cache timing attack on advanced encryption standard and countermeasures. Information and Automation for Sustainability (ICIAFs), 2010 5th International Conference on. December 2010, pp 177–182
37. Vattikonda BC, Das S, Shacham H (2011) Eliminating fine grained timers in Xen. In: Cachin C, Ristenpart T (eds) CCSW. ACM, New York, pp 41–46

Appendix A: CPUs Used for Experiments

The attacks and models developed in this book were tested on several systems. We selected a wide range of systems based on the technology used and the target application. The technology ranged from 22 to 90 nm and covered servers, desktops, laptops, and notebooks. Table A.1 has the details of the machines, the technology, clock, microarchitecture, and their cache configurations.

Table A.1 Cache configurations for test platforms

	Cache	Size	Line	Associativity
AMD Opteron (275)	*90 nm, 2.2 GHz, Italy*			
	L1-data	64 KB	64 byte	2-way
	L1-instruction	64 KB	64 byte	2-way
	L2-unified	1 MB	64 byte	16-way
Intel Xeon (E5345)	*65 nm, 2.33 GHz, Core*			
($\alpha = 255, \beta = 20$)	L1-data	32 KB	64 byte	8-way
	L1-instruction	32 KB	64 byte	8-way
	L2-unified	4 MB	64 byte	16-way
Intel Core 2 Duo (E7500)	*45 nm, 2.93 GHz, Wolfdale-3M*			
	L1-data	32 KB	64 byte	8-way
	L1-instruction	32 KB	64 byte	8-way
	L2-unified	3 MB	64 byte	12-way
Intel Atom (N455)	*45 nm, 1.66 GHz, Pineview*			
	L1-data	24 KB	64 byte	6-way
	L1-instruction	32 KB	64 byte	8-way
	L2-unified	512 KB	64 byte	8-way

C. Rebeiro et al., *Timing Channels in Cryptography,*
DOI 10.1007/978-3-319-12370-7

Table A.1 (continued)

	Cache	Size	Line	Associativity
Intel i3 (550)	*32 nm, 3.2 GHz, Clarkdale*			
	L1-data	32 KB	64 byte	8-way
	L1-instruction	32 KB	64 byte	4-way
	L2-unified	256 KB	64 byte	8-way
	L3-unified	4 MB	64 byte	16-way
Intel Dual Core (P6100)	*32 nm, 2 GHz, Arrandale*			
($\alpha = 180, \beta = 16$)	L1-data	32 KB	64 byte	8-way
	L1-instruction	32 KB	64 byte	4-way
	L2-unified	256 KB	64 byte	8-way
	L3-unified	3 MB	64 byte	12-way
Intel Xeon (E5606)	*32 nm, 2.13 GHz, Westmere*			
($\alpha = 193, \beta = 7.5$)	L1-data	32 KB	64 byte	8-way
	L1-instruction	32 KB	64 byte	4-way
	L2-unified	256 KB	64 byte	8-way
	L3-unified	8 MB	64 byte	16-way
AMD Opteron (6272)	*32 nm, 1.4 GHz, Interlagos*			
	L1-data	16 KB	64 byte	4-way
	L1-instruction	64 KB	64 byte	2-way
	L2-unified	2 MB	64 byte	16-way
	L3-unified	6 MB	64 byte	64-way
Intel i7 (4770)	*22 nm, 3.4 GHz, Haswell*			
	L1-data	32 KB	64 byte	8-way
	L1-instruction	32 KB	64 byte	8-way
	L2-unified	256 KB	64 byte	8-way
	L3-unified	8 MB	64 byte	16-way