Multiparty Messaging and Key Exchange Protocols

Created by George Kadianakis
with the supervision of Kenny Paterson

Abstract

One of the most socially valuable aspects of modern technology is how it enables people of all backgrounds to communicate with each other in lightning speeds. Even though the speed and performance of these communication tools can be called modern, the same cannot be said about their security. One such underdeveloped field in terms of security is the field of group messaging systems. Even though cryptography is an established field of science these days, we still haven't developed group messaging tools that are able to resist motivated adversaries. As part of this thesis, we perform a brief survey of the various group messaging applications that are currently in use and informally discuss their security properties. We look at how end-to-end security can be achieved through group key exchange protocols and present their security model in depth. We put special focus on insider attacks of group key exchange protocols and we also present a novel insider attack on a published protocol. We then discuss techniques for proving the security of a group key exchange against certain types of attacks. To demonstrate these techniques we slightly modify a published group key exchange and then prove it secure. Finally, this thesis includes Go source code that simulates a published group key exchange as a means to verify its correctness.

Contents

1 Prologue **5**

1.1 Background 5

1.2 Motivation 5

1.3 Contributions 6

1.4 Document Structure 6

2 Group Messaging Security **7**

2.1 Group Messaging Security Models 7

2.1.1 The Need for End-To-End Security 7

2.1.2 Off-The-Record Messaging 7

2.2 Further Security Properties for Group Messaging 8

2.3 Survey on Deployed Group Messaging Systems 8

2.4 Group Key Exchange and Group Messaging 9

2.5 Previous work on Group Messaging Security 10

3 Insider Attacks on Group Key Exchange **11**

3.1 Impersonation Attacks 11

3.2 Key Integrity Attacks 12

3.3 Key Control Attacks 12

4 Provable Security and Group Key Exchange **13**

4.1 Security Models for Group Key Exchange 13

4.1.1 History 13

4.1.2 The Execution Environment 14

4.2 Security Definitions 15

4.2.1 Authenticated Key Exchange Security 15

4.2.2 Mutual Authentication 16

4.2.3 Security Model and The Real World 17

4.3 Provable Security and Key Exchange 17

4.3.1 Computational Security Proofs for Key Exchange 18
4.3.2 Sequence of Games Proofs 18
4.3.3 Random Oracle Model 18

5 Analysis of an mBD+S Protocol **21**
5.1 Introduction 21
5.2 The mBD+S Protocol 21
5.3 Insider Attack on the mBD+S Protocol 22
5.4 Insider Attack Proof of Concept 23

6 Analysis of a Group Key Exchange Protocol **25**
6.1 Introduction 25
6.2 Protocol Overview 25
6.3 Rationale for a Modified Symmetric Protocol 26
6.4 Our Modified Symmetric Protocol 27
6.5 Security Proof for Modified Protocol 27

7 Conclusions and Future Directions **35**

A Mathematical Background **37**
A.1 Computational Assumptions 37
A.2 Crypto Background 37
A.2.1 Digital Signatures 38

B Implementation of a Deniable Group Key Agreement **41**

Chapter 1

Prologue

1.1 Background

Humans have a natural need to communicate with each other and with the recent advancements of technology, people's ability to communicate has reached new levels. It is easy and natural now for people to communicate in real-time with their peers remotely over computer screens.

The first *instant messaging protocols* were developed even before the Internet. Protocols like *Talkomatic*, which worked over the PLATO system, and *BITNET relay*, which worked over BITNET, saw significant use by universities during the 80s and influenced the way messaging protocols are designed now. After the rise of the Internet, hundreds of messaging protocols have been developed to suit the various use cases: from low-latency to high-latency, and from two-party chats down to thousands of people.

Nowadays, most Internet users use some sort of messaging protocol as part of their daily routine (e.g. Skype, IRC, Facebook chat, etc.), and as of 2014, companies like Skype serve *millions of users* who spend *billions of minutes* everyday conversing with each other [1] [2].

1.2 Motivation

Humans also have a natural need for privacy. Almost everyone expects some degree of privacy in their life and personal activities: Most people speak with their lovers in private, pull the curtains before changing clothes and shut the door of the toilet before using it. It is the author's opinion

[1] http://blogs.skype.com/2014/05/01/the-skype-community-welcomes-its-2000000th-user/

[2] http://blogs.skype.com/2013/04/03/thanks-for-making-skype-a-part-of-your-daily-lives-2-billion-minutes-a-day/

that it should be possible for people to experience their life with us much privacy as they choose to, and that this should also include their online activities.

Unfortunately, even though online messaging systems are so widely used, their privacy is nowhere close to robust. Most deployed chat systems leak information to third parties without making this obvious to the users. Information might be leaked to the local network, to advertisement firms, to the company that runs the chat system or to law enforcement.

1.3 Contributions

As part of this thesis, we provide a brief survey on group messaging protocols that have been developed. We put special focus on their security properties and their group key exchange.

More specifically, we examine insider attacks on such group key exchanges and their implications to group messaging. We present an insider attack of our own on a published group key exchange and look at security models that aim to cover such insider attacks.

We also look at techniques for proving group key exchange secure against various adversaries. Finally, we present a proof of our own against a slightly modified version of a published protocol.

1.4 Document Structure

The rest of the thesis is structured in the following manner:

- Chapter 2 is an introduction to group messaging systems and their security. We discuss various security properties that group messaging systems should have, and then examine a few deployed systems and their security. We introduce key exchange, by looking at how they can be used to provide end-to-end security in group messaging systems. Finally, we give a brief overview of previous research on group key exchange for group messaging applications.
- Chapter 3 provides informal descriptions of insider attacks on group key exchange, as well as their implications to the group messaging application.
- Chapter 4, motivated by the previous chapter, details the security model and security properties of group key exchanges. Finally, we give an introduction into proof techniques that could be used to prove such group key exchange secure.
- Chapter 5 presents a novel insider attack on the mBD+S protocol presented in *"Flexible group key exchange with on-demand computation of subgroup keys."* by Abdalla et al. [1]
- Chapter 6 provides a security proof for a slightly modified version of the protocol in *"Secure group key establishment revisited"* by Bohli et al. [7]

Chapter 2

Group Messaging Security

By "group messaging", we usually refer to text-based protocols that allow participants to exchange messages with each other. Such protocols can be synchronous and real-time as is the case for IRC, or asynchronous as is the case for most messaging systems on mobile phones (e.g. WhatsApp). This section focuses on the security of group messaging systems.

2.1 Group Messaging Security Models

Group messaging security is a new field of study and there is currently no widely accepted definition of what it means for a group messaging system to be secure.

2.1.1 The Need for End-To-End Security

Because of the lack of robust security models in this space, it's easy to claim that a group messaging system is secure and private, even if it provides no protection against realistic threats. For example, most widely used group messaging systems today are server-based and assume that the server is honest. In other words, they do not attempt to provide any *end-to-end security* whatsoever.

2.1.2 Off-The-Record Messaging

Fortunately, there has been research on security models for two-party instant messaging systems that we could use as a basis for defining security on the group setting.

One important security philosophy for two-party instant messaging is *Off-The-Record Messaging* as established by Borisov, Goldberg and Brewer [8]. While not a formal security model per se, it establishes important security properties that secure messaging systems should have.

Their idea is that instant messaging over the Internet should be able to mimic casual conversations of friends in a private space. That is, it must provide the following informal security properties:

End-to-end Confidentiality: No one apart from the two participants should be able to read the conversation.

Authentication: Participants are assured that their correspondents are who they think they are.

Forward Secrecy: Even if an adversary compromises the long-term keys of the participants, he should not be able to decrypt past encrypted communication

Deniability: Even though there is authentication, no participant should be able to cryptographically prove that this communication took place between those two parties to a third party.

Goldberg et al. implemented this philosophy into a protocol called *OTR*, which is a well-known example of usable secure messaging. However, OTR is a two-party protocol and tweaking it into a multiparty protocol is not straightforward.

2.2 Further Security Properties for Group Messaging

Multiparty messaging can also have security properties that are not applicable in the two-party model. For example:

Room Consistency: All participants should see the exact same room setup. An adversary should not be able to insert or remove participants from the group.

Transcript Consistency: All participants should see the exact same chat transcript. An adversary should not be able to reorder or hide messages from a subgroup of the participants.

Subgroup Messaging: It should be possible to efficiently send end-to-end secured messages to just a subgroup of the participants.

A more detailed survey on security properties that are applicable to messaging systems can be found on a recent paper by Unger et al. [31]

2.3 Survey on Deployed Group Messaging Systems

This section briefly examines some group messaging systems actually in use. There are not many *secure* instant messaging systems deployed currently, and we will see that most of the chat systems people use in their daily life are not actually end-to-end secure.

Skype, Facebook chat, etc.

Skype, Facebook, Google and other web platforms have their own web chat systems. Most of these web applications actually use XMPP [1] for communication, or implement their own closed-source messaging protocol. Currently, **none of these protocols are end-to-end**, which means that corporate servers can read all the messages or they can impersonate users in conversations.

IRC and XMPP

IRC [2] and XMPP MUCs have been widely used for group messaging. However, they are old protocols, not designed with security in mind and **they provide no end-to-end security**.

SILC

SILC [3] is one of the oldest attempts to provide secure group messaging. It uses its own chat framework and requires dedicated server infrastructure, in contrast to overlay protocols like OTR that can be used on top of already existing chat framework (e.g. IRC, XMPP). Also, the SILC protocol does not provide end-to-end security by default. Users can specify their own private key to encrypt their messages in an end-to-end fashion, however this approach does not provide proper perfect forward secrecy or authentication.

2.4 Group Key Exchange and Group Messaging

Since most of the above systems seem to suffer by end-to-end security issues, it would be appropriate to look deeper into how end-to-end security can be achieved. The standard way to achieve end-to-end security, is for the participants of a group messaging system to derive common cryptographic keys in such a manner that they cannot be guessed by an outsider attacker.

There are multiple ways that participants of a group can derive common keys. As an example, a group leader can be assigned who chooses a key and runs a *key transport protocol* to transfer the key to the other participants [16]. Or a *two-party key exchange* can be conducted pairwise between all n participants resulting in n^2 pairwise keys that can be used to encrypt messages to everyone as a group [27].

---

[1] http://xmpp.org/

[2] http://www.ietf.org/rfc/rfc1459.txt

[3] http://www.silcnet.org/

In this work, we focus on *group key exchange* and their role in group messaging. Specifically, we study *authenticated group key exchanges*, which allow authentication and key exchange to happen with a single run of the protocol.

Group key exchange are complicated cryptographic constructions and have various security properties. We believe that our lack of understanding of group key exchange plays an important role for the tiny usage of end-to-end encryption in modern communications. The rest of this document focuses on group key exchange and their security.

2.5 Previous work on Group Messaging Security

Several attempts have been made to construct secure group messaging systems and to design group key exchange protocols that would be suitable for this particular application.

The authors of OTR proposed a generalization of two-party OTR to the multiparty use case [20]. Their work mainly focused on establishing security properties and security models, but the actual cryptographic primitives that would be used were specified in an abstract manner. Matthew Van Gundy [22] followed up by proposing a specific 4-round authenticated deniable group key exchange based on [6].

Liu et al. [25] proposed a group messaging protocol which attempts to keep all the properties of OTR. Their group key exchange is based on the Burmester-Desmedt group key exchange with modifications to make it more dynamic, allowing members to join and drop out of the conversation with little overhead.

The recently proposed (n+1)sec protocol [18] utilizes the group key exchange from [1] to provide a deniable group key exchange. It also attempts to provide *transcript consistency* by attaching pointers to previous messages in every message.

Finally, the TextSecure messaging protocol was extended to the group setting by sending messages to each recipient using the two-party TextSecure protocol [27]. The group messaging protocol uses multicast encryption by broadcasting a single encrypted message to all participants and then sending the decryption key to each participant in a pairwise fashion.

Chapter 3

Insider Attacks on Group Key Exchange

Group Key Exchange (GKE) protocols allow a group of users to establish a shared session group key which can be further used to provide confidentiality or authentication in different group applications.

In the setting of key exchange, attacks that aim to undermine the security of the session key have been studied for years, both for the two-party case and the group setting (for a taxonomy of known key attacks see [29]). However, attacks that can be launched by *malicious protocol participants* are less well understood. We informally call such attacks *insider attacks*.

Malicious protocol participants can derive the session key by simply following the protocol, so learning the session key is not an interesting attack in this threat model. Instead, we are interested in attacks that allow insiders to impersonate other participants, or to disable other security properties of the key exchange. This section aims to present informal definitions of such attacks, as well as demostrate them in the context of group messaging.

3.1 Impersonation Attacks

Unknown Key Share Attack

The Unknown Key Share attack (or identity-misbinding attack) is a type of impersonation attack, first presented by Diffie et al. [17]. Informally speaking, if such an attack is mounted against Alice, then Alice believes to share a session key with Bob, whereas in fact Alice shares a key with a third party Eve. In most cases the attack can be launched by outsiders [15] but there are also cases where the attack can only be launched by malicious protocol participants [24].

Recently, Bader et al. [19] presented an Unknown Key Share attack that applies specifically to the group messaging protocol Textsecure along with a few possible exploitation scenarios.

Key Compromise Impersonation Attack

A Key Compromise Impersonation attack (KCI) is a type of impersonation attack that can be launched to Alice by Eve *only after compromising Alice's long-term key.* For example, if a protocol is vulnerable to a KCI attack and if Eve has compromised Alice's long-term private key, then Eve can successfuly pretend to be Bob when talking to Alice. This attack can be launched both by outsiders but also by protocol insiders as presented by Boyd et al [21].

It is worth mentioning that *KCI attacks can be particularly effective in the context of group messaging*: Imagine that the mafia needs to acquire some data from a corporate system. The mafia hackers manage to compromise the system only to find out that the the desired data has been moved elsewhere. However, it manages to compromise the long-term group messaging keys of that system. Leveraging a KCI attack in the next group messaging session, the mafia can impersonate a trusted confident of the company and request the data to be delivered to them or placed again on the compromised system.

3.2 Key Integrity Attacks

Key integrity attacks are attacks that force a subgroup of participants of a key exchange to derive different session keys from other participants. There are various examples of this attack on group key exchange [14], and we also present an attack of this kind on Section 5.

Key integrity attacks can also apply to group messaging: Imagine that members of the mafia have infiltrated a multi-stakeholder corporation. At some point, the corporation's executive board schedules an emergency meeting to notify the stakeholders of some important updates. If that meeting happened using a group messaging protocol that suffers from key integrity attacks, the mafia could force some participants to derive a different session key than others, effectively blocking the important information from getting to them.

3.3 Key Control Attacks

A key control attack can be launched by malicious protocol parcicipants to force the resulting session key to have a specific value, or to lie into a restricted set of values. It was first presented by Mitchell et al. [28] and since then various other protocols have been found to be vulnerable.

Key control attacks don't seem to be particularly effective against properly designed group messaging protocols. Section 6.3 details some possible attacks that could be launched against weakly designed group messaging protocols that use a key exchange suffering from key control issues.

Chapter 4

Provable Security and Group Key Exchange

Because of all these possible attacks, we want to be able to concretely reason about protocols and the security they offer. To do so, we have defined security models that consider the protocol participants, their trust relationship, communication environment and further relevant aspects, as well as definitions of required security goals. With such formalized models, we are able to examine the security of protocols with mathematics and logic and apply reductionist methods to prove protocols secure given certain assumptions.

4.1 Security Models for Group Key Exchange

A security proof of a group key exchange is only as secure and robust as its security model. Hence, we first take a look at published security models of group key exchange.

4.1.1 History

Over the past years, there have been many proposed security models for key exchange and some of them have also been adapted to work on the group setting. For a comprehensive survey see [26]. In this document, we use the security model presented in [21]. In the following paragraph, we provide a short timeline of its development.

While new protocols for key exchange and key distribution have been getting published since the 80s, there was no way to formally validate or prove such protocols secure. The security of such earlier group key exchange protocols has been analyzed with ad-hoc methods and heuristics based on informal definitions. Many such protocols were later found to be insecure, which urged cryptographers to develop systematic methods and frameworks to analyze and reason about key exchanges.

In 1993, Bellare and Rogaway presented an initial security model for two-party key exchange [3] loosely based on previous work by Bird et al. [5]. A few years later the same authors extended their

security model to also include the three-party case [4]. In 2000, Bellare. Pointcheval and Rogaway [2] further refined the way the model defines participant partnering and also gave corruption capabilities to the adversary allowing them to formally treat *forward secrecy*.

More recently, in 2001, Bresson, Chevassut, Pointcheval, and Quisquater [9] modified the previous two-party models to derive a security model designed for group key exchange protocols. That security model was further refined, using Shoup's concept of corruptions [30] by Bresson et al. [10, 11]. Finally, Boyd et al. [21] adapted those security models to also consider KCI attacks to arrive at the security model we will be using through this document.

4.1.2 The Execution Environment

Our security model puts the adversary $\mathcal{A}$ in the center of the universe. Informally speaking, the adversary completely controls the communication channel between the group participants and is able to create, edit, reorder or block any messages between them. She is also able to *corrupt* group participants compromising their short-term state or even their long-term key which effectively allows her to impersonate them. In this section, we clarify the adversary's capabilities and goals by defining a precise security model.

Protocol participants

We fix a non-emtpy set ID of n participants that want to take part in the group key exchange protocol P. Each participant $U_i \in ID$ can have multiple *instances* called *oracles*, involved in distinct concurrent executions of P. We denote instance s of player U_i as π_i^s with $s \in \mathbb{N}$.

Long-Lived Keys

Each player $U_i \in ID$ holds a long-lived key LL_U which can be either a pair of matching public/private keys or a symmetric key. LL_U is specific to U, not one of its instances. Associated to a protocol P is an LL-key generator G_{LL} which at initialization generates LL_U and assigns it to U.

Accepted state

An instance π_U^i enters an *accepted* state when it computes a session key sk_U^i. Note that an instance may terminate or abort without ever entering into an accepted state.

Partnering

Each protocol instance π_U^i of a participant is identified by a *unique session ID* sid_U^i. We assume that the session ID is generated during the protocol run.

Furthermore, each protocol instance π_U^i of a participant has a *partner ID* pid_U^i which is the set of identities of the participants with whom π_U^i wishes to establish a common group key. Note that pid_U^i includes the identity of U itself.

Two instances π_U^i and $\pi_{U'}^j$ at two different parties U and U' respectively are considered *partnered* if both instances have accepted, and $sid_U^i = sid_{U'}^j$ as well as $pid_U^i = pid_{U'}^i$.

Oracle queries

The adversary $\mathcal{A}$ interacts with the system by querying the participant *oracles*. The adversary is allowed to ask the following queries:

- Execute(sid, pid): Prompts a complete execution of the protocol among the parties in *pid* using the unique session ID *sid*. $\mathcal{A}$ is given all the protocol messages, modelling passive attacks.
- Send(π_U^i, m): Sends a message m to the instance π_U^i. If the message is (sid, pid) the instance π_U^i is initiated with session ID *sid* and partner ID *pid*. The response of π_U^i to any *Send* query is returned to $\mathcal{A}$.
- RevealKey(π_U^i): If π_U^i has accepted, $\mathcal{A}$ is given the session key sk_U^i established at π_U^i.
- Corrupt(U): The long-term secret key SK_U of U is returned to $\mathcal{A}$. This query returns neither the short-term session key (if computed) nor the internal state.
- RevealState(π_U^i): The internal state of U is returned to $\mathcal{A}$. We assume that internal state is erased once π_U^i has accepted. Hence, a *RevealState* query to an accepted instance returns nothing.
- Test(π_U^i): A random bit b is secretly chosen. If $b = 1$, $\mathcal{A}$ is given sk_U^i established at π_U^i. Otherwise, a random value chosen from the session key probability distribution is given. Note that a *Test* query is allowed only on an accepted instance.

4.2 Security Definitions

4.2.1 Authenticated Key Exchange Security

One of the basic security definitions of GKE protocols is *Authenticated Key Exchange security* (*AKE-security*) which ensures that the derived group key is indistinguishable from a random string to the adversary. This implies that the adversary has no better method of deriving the group key (or parts of it) than simply guessing it.

We first need to define the notion of *freshness* which is central to the definition of *AKE-security*. Informally, a session is considered fresh if the session key is not trivially compromised; hence the session can be challenged using a *Test* query:

Definition 1. *(Freshness) An instance π_U^i is considered* ***fresh*** *if the following conditions hold:*

- *the instance π_U^i or any of its partners has not been asked a* RevealKey *query after their acceptance.*
- *the instance π_U^i or any of its partners has not been asked a* RevealState *query before their acceptance.*
- *if π_U^j is a partner of π_U^i and $\mathcal{A}$ asked $Corrupt(U')$, then any message that $\mathcal{A}$ sends to π_U^i on behalf of π_U^j, must come from π_U^j intended to π_U^i.*

We can now give the following definition of *Authenticated Key Exchange security*:

Definition 2. *(AKE-security) An adversary $\mathcal{A}_{ake}$ against the AKE-security notion is allowed to make* Execute, Send, RevealState, RevealKey *and* Corrupt *queries in Stage 1. $\mathcal{A}_{ake}$ makes a Test query to an instance π_U^i at the end of Stage 1 and is given a challenge key K_b as described above. $\mathcal{A}_{ake}$ can continued asking queries during Stage 2. Finally, $\mathcal{A}_{ake}$ outputs a bit b' and wins the AKE security game if (1) $b' = b$* ***and*** *(2) if the instance π_U^i that was asked the Test query remained fresh till the end of $\mathcal{A}_{ake}$'s execution. Let $Succ_{\mathcal{A}_{ake}}$ be the success probability of $\mathcal{A}_{ake}$ in winning the AKE security game. The advantage of $\mathcal{A}_{ake}$ in winning this game is $Adv_{\mathcal{A}_{ake}} = |2Pr[Succ_{\mathcal{A}_{ake}}] - 1|$.*

A protocol is called AKE-secure if $Adv_{\mathcal{A}_{ake}}$ is negligible given a security parameter k for any probabilistic polynomial-time algorithm $\mathcal{A}_{ake}$.

In our security model, the definition of *AKE-security* and *freshness* are specially designed to also take *KCI attacks* on account. In an implicit authentication protocol, an outsider who manages to derive the session key of another participant A, can impersonate A to anyone else. The third *freshness* condition requires that the adversary remains passive for any partner that it corrupts. That's natural, since *AKE-security is an outsider security notion*, and an adversary that impersonates corrupted parties would be considered an insider. Furthermore, our definitions imply *forward secrecy* as $\mathcal{A}_{ake}$ is allowed to obtain the long-term private keys of all parties, without being able to break the AKE security of old sessions.

4.2.2 Mutual Authentication

Since the AKE security definition was designed to handle attacks by outsiders, it's important to also define a security definition that captures attacks by malicious insiders. For this reason we define the *Mutual Authentication* notion that models impersonation attacks and ensures agreement on the session key in the presence of insiders.

Definition 3. *(Mutual Authentication) An adversary $\mathcal{A}_{ma}$ against the mutual authentication notion of a GKE protocols is allowed to make* Execute, Send, RevealState, RevealKey *and* Corrupt *queries. $\mathcal{A}_{ma}$ violates the mutual authentication property of the protocol if at some point during the protocol*

run, there exists an uncorrupted instance π_U^i (although the party U may be corrupted) that has accepted a key sk_U^i and another party $U' \in pid_U^i$ that is uncorrupted at the time π_U^i accepts such that:

1. *there is no instance $\pi_{U'}^j$ with $(pid_{U'}^j, sid_{U'}^j) = (pid_U^i, sid_U^i)$* ***or***
2. *there is no instance $\pi_{U'}^j$ with $(pid_{U'}^j, sid_{U'}^j) = (pid_U^i, sid_U^i)$ that has accepted with $sk_{U'}^j \neq sk_U^i$.*

Let $Succ_{\mathcal{A}_{ma}}$ be the success probability of $\mathcal{A}_{ma}$ in winning the mutual authentication game.

A protocol is said to provide mutual authentication in the presence of insiders if $Succ_{\mathcal{A}_{ma}}$ is negligible given a security parameter k for any polynomial time $\mathcal{A}_{ma}$.

The MA definition in our security model is stronger than earlier definitions, since it captures the additional goal of KCI in the presence of insiders by giving the adversary the extra power of corrupting the long-term private key of participants.

4.2.3 Security Model and The Real World

It is easy to see that a crucial step in analyzing the security of a cryptographic protocol is defining a suitable security model. The appropriateness of the security model depends on how well it captures adversarial actions in the real world.

By allowing the adversary to issue queries to the oracles, we model information leakage by various real world attacks. For example, the Corrupt query models the compromise of long-term private keys which could happen when a hacker compromises a system. Similarly, the RevealKey query models an attack where the hacker breaks into the system right after the key exchange, just in time to steal the resulting session key but not the victim's long-term key.

By giving the adversary access to all those oracles we let him combine all those real world attacks in an attempt to defeat the stated security definitions. The security definitions represent concrete flaws of the key exchange, hence *if no possible adversary* can defeat them, we can say that the key exchange protocol is secure.

4.3 Provable Security and Key Exchange

Modern cryptography employs *provable security* to reason about the security of cryptographic protocols. That is, a cryptographic construction provides certain security goals if there exists a corresponding proof of security in some mathematically indisputable way. In this thesis, we focus on proof techniques that rely on computational complexity.

4.3.1 Computational Security Proofs for Key Exchange

The core idea of computational security proofs is that all involved parties (including the adversary) are modeled by probabilistic polynomial-time algorithms whose outputs follow some probability distribution. In such security proofs, we usually model each security goal as an adversarial attack that the adversary can launch by interacting with the honest parties. In such proofs, the adversary is considered to be successful if a certain event *Win* occurs with a non-negligible probability.

Security proofs in computational security models are mostly of *reductionist* nature and carried out by contradiction. The idea behind the reduction is usually to show is to show that some intractable problem P_1 implies the hardness of a problem P_2 which was specially constructed to guarantee a specific security goal. For this purpose, it is usually shown that a solver algorithm for P_2 can also solve the intractable P_1 with polynomially bounded additional complexity.

4.3.2 Sequence of Games Proofs

Modern key exchange protocols rely on multiple cryptographic primitives as building blocks. Constructing reductions proofs of such protocols is quite complex and error-prone. A technique called "sequence of games" was developed to reduce complexity of such reductionist proofs.

In the "sequence of games" approach one constructs a sequence of games $G_0, G_1, \ldots, G_n$ starting with G_0 being the original security model. The original game, specifies an event *Win* that must occur in case of a successful attack in the original protocol. Subsequent games define more events Win_i that are all related to Win. Then one tries to show that the probability of Win_i is negligible close to that of Win_{i+1}. Finally, the probability of the adversary to win the final game Win_n should be *equal* to the target probability according to the security goal definition in the underlying security model. By creating this chain of negligible events, leading to a final fair game, we know that also in the original game the adversary had only negligible chance of winning.

4.3.3 Random Oracle Model

One non-standard methodology frequently applied to computational security proofs is the *Random Oracle Model*. The random oracle model involves modelling certain parts of cryptosystems, called *hash functions*, as totally random functions about whose internal workings the attacker has no information.

This is useful because hash functions are problematic from a provable security point of view. This is because there are various ways to construct them and they are commonly developed using symmetric techniques which are difficult to prove using reductionist logic since there are no natural intractable problems to reduce them to. The random oracle model attempts to overcome our inability to make strong statements about the security of hash functions by modelling them as completely

random functions about which an attacker has no information. A "sequence of games" proof that uses the random oracle model will be demonstrated on section 6

It is worth mentioning that the random oracle model is considered controversial in provable security. This is because it allows cryptographers to take shortcuts that simply can't be supported in the real world we live in. One important result in this area is from Canetti et al. [13] who demonstrated that it was possible to have a scheme that was provably secure in the random oracle model, and yet insecure when the random oracle was replaced with *any hash function*.

Chapter 5

Analysis of an mBD+S Protocol

5.1 Introduction

In 2010, Abdalla et al. [1] proposed two group key exchange protocols: *mBD+P* and *mBD+S* based on the group key exchange protocol by Burmester and Desmedt [12].

The original paper proved the AKE-security of those protocols in the random oracle model. However, the protocols were found vulnerable by Cheng et al. [14] to a key integrity attack that could be launched by malicious insiders. While this was not a violation of AKE security, we believe insider attacks are important and secure group key exchange should be designed appropriately.

In this section we review the *mBD+S* protocol and then demonstrate a novel key integrity attack that can be launched by malicious insiders on the protocol. Our attack is similar to the issue that Bohli et al. [7] presented on the Kat and Yung protocol [23].

5.2 The mBD+S Protocol

The *mBD+S* protocol works in two stages: the group stage and the subgroup stage.

The group stage of the mBD+S protocol is run initially amongst all participants to derive a common group key. Then, any subset of participants can run the subgroup stage on-demand to derive a private subgroup key.

The group stage is two rounds whereas the subgroup stage is one round, hence running the subgroup stage is cheaper than running a separate group stage for the subgroup. In this chapter we focus on the group stage of the protocol.

Notation

Through this chapter we use the notation of the original paper. That is, by $\mathbb{G}$ we denote a cyclic group of prime order generated by a group element g. By $H, H_g : \{0,1\}^* \to \{0,1\}^k$ we denote two cryptographic hash functions. The symbol "|" will be used for the concatenation of bit-strings. Since the protocol is authenticated, we will make use of a digital signature scheme $\Sigma = (Keygen, Sign, Verify)$ which we assume to be existentially unforgeable under chosen message attacks (UF-CMA).

Let the group of users be defined by $pid = (U_1, \ldots, U_n)$. In the following description we assume that user indices form a cycle such that $U_0 = U_n$ and $U_{n+1} = U_1$. Figure 5-1 outlines the execution of the protocol after initialization.

Protocol

Round 1

Computation Each U_i computes $y_i = g^{x_i}$ for some random $x_i \in_R \mathbb{Z}_q$.

Broadcast Each U_i broadcasts (U_i, y_i).

Round 2

Computation

1. Each U_i sets $sid_i = (U_1|y_1, \ldots, U_n|y_n)$.
2. Each U_i sets $k'_{i-1,i} = y_{i-1}^{x_i}$ and $k'_{i,i+1} = y_{i+1}^{x_i}$
3. Each U_i sets $z'_{i-1,i} = H(k_{\ i-1,i}, sid_i)$ and $z'_{i,i+1} = H(k'_{i,i+1}, sid_i)$
4. Each U_i sets $z_i = z'_{i-1,i} \oplus z'_{i,i+1}$
5. Each U_i sets $\sigma_i = Sign(sk_i, (U_i, z_i, sid_i))$

Broadcast Each U_i broadcasts (U_i, z_i, σ_i)

Check

Each U_i checks that $z_1 \oplus \ldots \oplus z_n = 0$ and whether all received signatures σ_j are valid and aborts if any of these checks fails.

Group Key Computation

1. Each U_i iteratively for each $j = i, \ldots, i+n-1$ computes $z'_{j,j+1} = z'_{j-1,j} \oplus z_j$
2. Each U_i accepts $k_i = H_g(z'_{1,2}, \ldots, z'_{n,1}, sid_i)$ as the group session key.

Figure 5-1: The mBD+S protocol from [1]

5.3 Insider Attack on the mBD+S Protocol

We now present an attack that can be launched by malicious insiders and aims to *disturb the key integrity* of the protocol. In this attack, the goal of the adversary $\mathcal{A}$ is to force some involved participants to compute a different session key than the rest, while having the same session identifier.

Say $n > 3$, then the adversary $\mathcal{A}$ can mount the following attack in a new protocol session:

1. $\mathcal{A}$ *corrupts* U_1 and U_3. Since $\mathcal{A}$ has corrupted U_1 and U_3 she can compute signatures on their behalf and impersonate them completely.

2. In the second protocol round, $\mathcal{A}$ computes z_1 and z_3 as specified, and then sets $\overline{z_1} = z_3$ and $\overline{z_3} = z_1$. $\mathcal{A}$ computes $\overline{\sigma_1} = Sign(sk_1, (U_1, \overline{z_1}, sid_1))$ instead of $\sigma_1 = Sign(sk_1, (U_1, z_1, sid_1))$, and similarly computes $\overline{\sigma_3} = Sign(sk_3, (U_3, \overline{z_3}, sid_3))$ instead of $\sigma_3 = Sign(sk_3, (U_3, z_3, sid_3))$. Now $\mathcal{A}$ broadcasts $(U_1, \overline{z_1}, \overline{\sigma_1})$ on behalf of U_1 and $(U_3, \overline{z_3}, \overline{\sigma_3})$ on behalf of U_3. In other words, $\mathcal{A}$ swaps the z_i-contributions of U_1 and U_3.

Now all protocol participants have computed the same sid_i, all of them have received the same messages but a subgroup of the participants will derive different session keys than the rest.

5.4 Insider Attack Proof of Concept

We show the attack on a group of 4 participants $U_1, \ldots, U_4$. Assuming that the attacker launched the above attack by corrupting U_1 and U_3, we now execute the *Check* and *Group Key Computation* phases to compute the derived session group key for honest participants U_2 and U_3 and demonstrate that they are different. Remember that the attacker has set $z_1 \leftarrow z'_{2,3} \oplus z'_{3,4}$ and $z_3 \leftarrow z'_{4,1} \oplus z'_{1,2}$

Check phase

First of all, we can see that the adversary can pass the *Check* phase since all the signatures are valid and furthermore $z_1 \oplus z_2 \oplus z_3 \oplus z_4 = 0$ stands since it expands to:

$$(z'_{2,3} \oplus z'_{3,4}) \oplus (z'_{1,2} \oplus z'_{2,3}) \oplus (z'_{4,1} \oplus z'_{1,2}) \oplus (z'_{3,4} \oplus z'_{1,4}) = 0$$

Group Key Computation phase

Next is the *Group Key Computation* phase, where we compute the group key as $k_i = H_g(z'_{1,2}, z'_{2,3}, z'_{3,4}, z'_{4,1}, sid_i)$. It is worth noting that during the second round of the protocol, U_2 had already derived $z'_{1,2}$ and $z'_{2,3}$, and U_4 has already derived $z'_{3,4}$ and $z'_{4,1}$. The table below details the computations of U_2 and U_4 when deriving their unknown $z'_{i,i+1}$ values:

Point of View of U_2	Point of View of U_4
$z'_{1,2}$ was derived in second round by U_2	$z'_{1,2} = z'_{4,1} \oplus z_1$ $= z'_{4,1} \oplus z'_{2,3} \oplus z'_{3,4}$
$z'_{2,3}$ was derived in second round by U_2	$z'_{2,3} = z'_{1,2} \oplus z_2$ $= z'_{4,1} \oplus z'_{2,3} \oplus z'_{3,4} \oplus z'_{1,2} \oplus z'_{2,3}$ $= z'_{3,4} \oplus z'_{4,1} \oplus z'_{1,2}$
$z'_{3,4} = z'_{2,3} \oplus z_3$ $= z'_{2,3} \oplus z'_{4,1} \oplus z'_{1,2}$	$z'_{3,4}$ was derived in second round by U_4
$z'_{4,1} = z'_{3,4} \oplus z_4$ $= z'_{2,3} \oplus z'_{4,1} \oplus z'_{1,2} \oplus z'_{3,4} \oplus z'_{1,4}$ $= z'_{1,2} \oplus z'_{2,3} \oplus z'_{3,4}$	$z'_{4,1}$ was derived in second round by U_4

Looking at the above table, we can now calculate the derived group keys of U_2 and U_4 using the formula

$$k_i = H_g(z'_{1,2}, z'_{2,3}, z'_{3,4}, z'_{4,1}, sid_i)$$

In our example, this gives us:

$$k_2 = H_g(z'_{1,2}, z'_{2,3}, z'_{2,3} \oplus z'_{4,1} \oplus z'_{1,2}, z'_{1,2} \oplus z'_{2,3} \oplus z'_{3,4}, sid_2)$$
$$k_4 = H_g(z'_{4,1} \oplus z'_{2,3} \oplus z'_{3,4}, z'_{3,4} \oplus z'_{4,1} \oplus z'_{1,2}, z'_{3,4}, z'_{4,1}, sid_4)$$

where $sid_2 = sid_4$ but all the other elements are unequal. Hence $k_2 \neq k_4$.

Chapter 6

Analysis of a Group Key Exchange Protocol

6.1 Introduction

In 2007, Bohli et al. [7] proposed a two-round group key exchange protocol and proved it secure for their definitions of outsider and insider security. Later, Boyd et al. [21] showed a new proof of security in the random oracle model.

In this chapter, we present the protocol and then modify it slightly to make it symmetric and easier to implement. Finally, we prove the modified protocol secure in the random oracle model against the AKE security definition.

6.2 Protocol Overview

Let $U_1, \ldots, U_n$ be the set of parties who wish to establish a common group key. It is assumed that the parties are ordered in a logical ring with U_{i-1} and U_{i+1} being the left and right neighbours of U_i for $1 \leq i \leq n$, $U_0 = U_n$ and $U_{n+1} = U_1$. During the initialization phase, a cyclic group $\mathbb{G}$ of prime order q, a generator g of $\mathbb{G}$ and the description of a hash function H that maps to $\{0,1\}^k$ are chosen. Each party is assumed to have a long-term private and public key pair for a public key signature scheme. Figure 6-1 on page 26 outlines the execution of the protocol after initialization.

It is worth noting that in this protocol all participants U_i do identical actions, except from participant U_n who does different actions. This asymmetry of the protocol should not influence its security and should not give U_n any advantage under the examined security model. On this note, Boyd et al. proved the protocol *AKE-secure*, and they also showed that it satisfies the *mutual authentication* and *contributiveness* security properties.

Round 1

Computation

1. Each U_i except U_n chooses $k_i \in_R \{0,1\}^k$, $x_i \in_R \mathbb{Z}_q$ and computes $y_i = g^{x_i}$. U_n additionally computes $H(k_n)$.
2. Each U_i except U_n sets $M_i^I = k_i \| y_i$, while U_n sets $M_n^I = H(k_n) \| y_n$.
3. Each U_i computes a signature σ_i^I on $M_i^I \| pid_i$.

Broadcast Each U_i broadcasts $M_i^I \| \sigma_i^I$.

Check Each U_i checks all signatures σ_j^I of incoming messages $M_j^I \| \sigma_j^I$ for $j \neq i$.

Round 2

Computation

1. Each U_i computes $t_i^L = H(y_{i-1}^{x_i})$, $t_i^R = H(y_{i+1}^{x_i})$, $T_i = t_i^L \oplus t_i^R$ and $sid_i = H(pid \| k_1 \| \ldots \| k_{n-1} \| H(k_n))$. U_n additionally computes $mask_n = k_n \oplus t_n^R$.
2. Each U_i except U_n sets $M_i^{II} = T_i \| sid_i$ while U_n sets $M_n^{II} = mask_n \| T_n \| sid_n$
3. Each U_i computes a signature σ_i^{II} on M_i^{II}.

Broadcast Each U_i broadcasts $M_i^{II} \| \sigma_i^{II}$.

Check

1. Each U_i verifies the incoming the signatures σ_j^{II} on the corresponding message M_j^{II} for every $j \neq i$. Each U_i also checks that $T_1 \oplus \ldots \oplus T_n \stackrel{?}{=} 0$ and $sid_i \stackrel{?}{=} sid_j$.
2. Each U_i for $i < n$, extracts $k_n = mask_n \oplus T_1 \oplus \ldots \oplus T_{i-1} \oplus t_i^L$ and checks the commitment $H(k_n)$ sent in Round 1 for the k_n extracted.

Key Computation

Each U_i computes the session key $sk_i = H(pid_i \| k_1 \| \ldots \| k_n)$

Figure 6-1: Original protocol from [7]

6.3 Rationale for a Modified Symmetric Protocol

In this section we present a slightly modified version of the protocol in Figure 6-1 to make it symmetric. While no security flaws have been found in the protocol or in its security proofs, we think that symmetric protocols are more elegant and they are also easier to implement since less code needs to be written and no special procedures are needed for choosing the distinguished participant.

Furthermore, the assymetric protocol has a slightly weaker notion of key control, since the distinguished participant can wait until all the other participants have sent their k_i contributions in the first round, which allows her to predetermine some bits of the session key by carefully choosing her k_n contribution. While the distinguished participant cannot set the whole session key to a specific value of her choosing, she can still predetermine b bits of the session key by brute forcing through 2^b possible k_n values.

The above issue is not covered under the usual contributiveness definition, and hence the original protocol was proved secure under contributiveness by Boyd et al. It is also worth noting that this shortcoming is not severe in the case where this group key exchange is used in a group messaging

application assuming that the resulting secret is used carefully to achieve end-to-end security. An example of how this can go wrong would be if the group messaging protocol splits the session key into parts and uses those parts individually to achieve security. In that case, and if the parts are sufficiently small, the adversary could set those bits to specific values which might be able to influence security. However, these attacks are theoretical enough that we don't consider them serious security threats.

Finally, this modification was also suggested in the original paper by Bohli et al. but they never demonstrated the modified protocol or proved it secure.

6.4 Our Modified Symmetric Protocol

In this section we present our modified symmetric protocol. The initialization phase is the same as the one described in Section 6.2, and the rest of the protocol can be seen in Figure 6-2 on page 28.

The main difference between our modified protocol and the original protocol, is that in our modified version all participants *commit* to their key contribution k_i in the first round, and publish their contributions encrypted with the Burmester-Desmedt keys on the second round. In the final step, participants decrypt all the encrypted key contributions they have received and use them to calculate the group key.

The above procedure is also demonstrated in the deniable key exchange protocol by Bohli et al. [6], but their protocol requires four rounds to complete because of the deniability requirement.

6.5 Security Proof for Modified Protocol

We now show that our modified protocol in Figure 6-2 is AKE-secure as per Definition 2. Our proof required minimal modifications to the proofs of the original protocol [7, 21].

Theorem 6.5.1. *The protocol in Figure 6-2 is AKE-secure as per Definition 2 assuming that the CDH assumption holds in $\mathbb{G}$, the signature scheme is UF-CMA secure and that H is a random oracle. The advantage of $\mathcal{A}$ is upper bounded by:*

$$Adv_{\mathcal{A}_{ake}} \leq 2\left(n^2 Adv^{cma} + \frac{((n+2)q_s + q_r)^2}{2^k} + \frac{q_s^2}{2^k} + nq_s q_r Succ^{cdh} + \frac{q_s q_r}{2^{kn}}\right)$$

where n is the number of participants, Adv^{cma} is the advantage of a polynomial adversary against the UF-CMA security of the signature scheme, $Succ^{cdh}$ is the probability of solving CDH in $\mathbb{G}$ and k is the security parameter. q_s and q_r are the upper bounds on the number of Send *and random oracle queries respectively that $\mathcal{A}_{ake}$ can ask.*

Proof. We provide the proof in a sequence of games. Let S_i be the event that $\mathcal{A}_{ake}$ wins the AKE-

Round 1

Computation

1. Each U_i chooses $k_i \in_R \{0,1\}^k$, $x_i \in_R \mathbb{Z}_q$ and computes $y_i = g^{x_i}$.
2. $\boxed{\text{Each } U_i \text{ sets } M_i^I = H(k_i)\|y_i.}$
3. Each U_i computes a signature σ_i^I on $M_i^I\|pid_i$.

Broadcast Each U_i broadcasts $M_i^I\|\sigma_i^I$.

Check Each U_i checks all signatures σ_j^I of incoming messages $M_j^I\|\sigma_j^I$ for $j \neq i$.

Round 2

Computation

1. Each U_i computes $t_i^L = H(y_{i-1}^{x_i})$, $t_i^R = H(y_{i+1}^{x_i})$, $T_i = t_i^L \oplus t_i^R$ and $\boxed{sid_i = H(pid\|H(k_1)\|\ldots\|H(k_{n-1})\|H(k_n)).}$
2. $\boxed{\text{Each } U_i \text{ encrypts their } k_i \text{ contribution as } ek_i = k_i \oplus t_i^R.}$
3. $\boxed{\text{Each } U_i \text{ sets } M_i^{II} = ek_i\|T_i\|sid_i.}$
4. Each U_i computes a signature σ_i^{II} on M_i^{II}.

Broadcast Each U_i broadcasts $M_i^{II}\|\sigma_i^{II}$.

Check

1. Each U_i verifies the incoming the signatures σ_j^{II} on the corresponding message M_j^{II} for every $j \neq i$.
2. Each U_i also checks that $T_1 \oplus \ldots \oplus T_n \stackrel{?}{=} 0$ and $sid_i \stackrel{?}{=} sid_j$.
3. $\boxed{\text{Each } U_i \text{ extracts } k_j = ek_j \oplus T_{j+1} \oplus \ldots \oplus T_{i-1} \oplus t_i^L \text{ for every } j \neq i.}$
4. $\boxed{\text{Each } U_i \text{ checks the commitment } H(k_j) \text{ sent in Round 1 for the } k_j \text{ extracted.}}$

Key Computation

Each U_i computes the session key $\boxed{sk_i = H_g(pid_i\|k_1\|\ldots\|k_n).}$

Figure 6-2: Our modified protocol

security game in Game i. We also define τ_i to be the advantage of $\mathcal{A}_{ake}$ in Game i computed as

$$\tau_i = |2Pr[S_i] - 1| \tag{6.1}$$

Now we present the sequence of games:

Game 0: In this first game, the protocol participants' oracles are faithfully simulated for the adversary i.e. the system behaves as in the real model. By definition we have:

$$Adv_{\mathcal{A}_{ake}} = \tau_0 = |2Pr[S_0] - 1| \tag{6.2}$$

Game 1: This is the same game as the previous game, except that the simulation fails if an event **Forge** occurs. Hence:

$$|Pr[S_1] - Pr[S_0]| \leq Pr[Forge]$$

This allows us to update equation 6.2 as follows:

$$\begin{aligned} Adv_{A_{ake}} = \tau_0 &= |2Pr[S_0] - 1| = |2Pr[S_0] - 2Pr[S_1] + 2Pr[S_1] - 1| \\ &\leq |2Pr[S_0] - 2Pr[S_1]| + |2Pr[S_1] - 1| \\ &\leq 2Pr[Forge] + \tau_1 \end{aligned} \tag{6.3}$$

Informally, the event **Forge** occurs when $\mathcal{A}_{ake}$ manages to forge a signature of an uncorrupted participant. It occurs when $\mathcal{A}_{ake}$ issues a *Send* query with a message of the form (M_i, σ_i) such that U_i is not corrupted and the message has not been an output of an instance at U_i.

If such a signature forgery occurs as part of $\mathcal{A}_{ake}$, we can use $\mathcal{A}_{ake}$ to construct a forger $\mathcal{F}$ which outputs a forgery (σ, m) with respect to any given public key K_p:

Our forger $\mathcal{F}$ receives as input the target public key K_p and access to a public signing oracle. $\mathcal{F}$ will emulate the whole system and use $\mathcal{A}_{ake}$ as a subroutine for executing the forging attack.

To start, $\mathcal{F}$ initializes all participant keys as normal according to the protocol, except from U_i where $\mathcal{F}$ sets its public key to K_p. $\mathcal{F}$ then runs $\mathcal{A}_{ake}$ as a subroutine and simulates the participants by answering oracle queries made by $\mathcal{A}_{ake}$ as follows:

When $\mathcal{A}_{ake}$ makes a *Send* query, $\mathcal{F}$ answers in a straightforward way, using the long-term keys of the oracles to sign the flows, except when the query is towards an oracle of U_i, in which case $\mathcal{F}$ needs to use the signing oracle to derive a valid signature. When $\mathcal{A}_{ake}$ makes a *Corrupt*-query, $\mathcal{F}$ again answers in a straightforward way except if the target is U_i, in which case $\mathcal{F}$ does not know the long-term key and hence stops and fails.

If $\mathcal{A}_{ake}$ makes a query of the form $Send(*, (\sigma, m))$ where σ is a valid signature on m with respect to K_p and (σ, m) have not been encountered before, then $\mathcal{F}$ halts and outputs (σ, m) as a forgery.

The probability of $\mathcal{A}_{ake}$ not corrupting the wanted party is $\geq 1/n$. The probability of $\mathcal{A}_{ake}$ outputting a valid forgery on behalf of this user is also $\geq 1/n$. Hence the advantage of a chosen message attacker on this digital signature scheme is $Adv^{cma} \geq 1/n^2 Pr[Forge]$, which gives us:

$$Pr[Forge] \leq n^2 * Adv^{cma}$$

Adv^{cma} is negligible by the UF-CMA assumption, hence the event *Forge* occurs with negligible probability only.

Game 2: This game is identical to the previous game, but the simulation fails if an event **Collision** occurs. So for this game:

$$|Pr[S_2] - Pr[S_1]| \leq Pr[Collision]$$

Acting as before we get:

$$\begin{aligned} \tau_1 &= |2Pr[S_1] - 1| = \\ &\leq |2Pr[S_1] - 2Pr[S_2]| + |2Pr[S_2] - 1| \\ &\leq 2Pr[Collision] + \tau_2 \end{aligned} \tag{6.4}$$

The event *Collision* occurs when the random oracle H produces a collision for any of its inputs. Each *Send* query requires at most $n + 2$ queries to the random oracle, and the adversary can also query the random oracle directly. Hence, the total number of random oracle queries is bounded by $((n + 2)q_s + q_r)$, where q_s and q_r are the maximum number of *Send* and random oracle queries respectively that $\mathcal{A}_{ake}$ can ask. Hence, by the birthday paradox, the probability of *Collision* is:

$$Pr[Collision] \leq \frac{((n + 2)q_s + q_r)^2}{2^k}$$

Game 3: This game is identical to the previous game, but the simulation fails if an event **Repeat** occurs. For this game:

$$|Pr[S_3] - Pr[S_2]| \leq Pr[Repeat]$$

and the advantage of the adversary on this game is:

$$\begin{aligned}\tau_2 &= |2Pr[S_2] - 1| = \\ &\leq |2Pr[S_2] - 2Pr[S_3]| + |2Pr[S_3] - 1| \\ &\leq 2Pr[Repeat] + \tau_3\end{aligned} \tag{6.5}$$

The event *Repeat* occurs when an instance at a party U_i chooses a key contribution k_i that was previously chosen by an oracle of a participant. As there are a maximum q_s instances that may have chosen a nonce k_i, the birthday paradox gives us:

$$Pr[Repeat] \leq \frac{q_s^2}{2^k}$$

Game 4: This game is identical to the previous game except that the following rule is added: The simulator chooses a random value $\overline{s} \in_R [1, q_s]$, where q_s is the maximum number of protocol sessions that the adversary can activate. The value $\overline{s}$ represents a guess as to the protocol session in which $\mathcal{A}_{ake}$ is going to call the *Test* query. If $\mathcal{A}_{ake}$ does not ask the *Test* query during session $\overline{s}$, then the game is aborted.

Let's call *Guess* the event were the simulator guesses correctly. Then we have:

$$\begin{aligned}Pr[S_4] &= Pr[S_4 \wedge Guess] + Pr[S_4 \wedge \neg Guess] \\ &= Pr[S_4|Guess]Pr[Guess] + Pr[S_4|\neg Guess]Pr[\neg Guess]\end{aligned} \tag{6.6}$$

where $Pr[S_4|Guess]$ is the probability of winning provided that the guess was correct; since the adversary won't notice a difference if the guess is correct, its probability of winning Game 4 if the guess is correct is the same as winning in the previous game, that is $Pr[S_4|Guess] = Pr[S_3]$. Also, since $\overline{s}$ was picked randomly, the probability of guessing the *Test* session correctly is $Pr[Guess] = 1/q_s$. Finally, if the guess was incorrect, the simulator aborts by setting the bit b' at random, hence the probability that $\mathcal{A}$ will win in this case is $Pr[S_4|\neg Guess] = 1/2$. Using the above, we can continue working on equation 6.6:

$$\begin{aligned}Pr[S_4] &= Pr[S_4|Guess]Pr[Guess] + Pr[S_4|\neg Guess]Pr[\neg Guess] \\ &= Pr[S_3]\frac{1}{q_s} + \frac{1}{2}(1 - \frac{1}{q_s})\end{aligned} \tag{6.7}$$

Hence from equations 6.1 and 6.7 we have:

$$\begin{aligned}\tau_4 &= |2Pr[S_4] - 1| \\ &= \left|2(Pr[S_3]\frac{1}{q_s} + \frac{1}{2}(1 - \frac{1}{q_s})) - 1\right| \\ &= \left|\frac{2Pr[S_3] - 1}{q_s}\right| \\ &= \frac{\tau_3}{q_s}\end{aligned}$$

which gives us:

$$\tau_4 = \frac{1}{q_s}\tau_3 \implies \tau_3 = q_s\tau_4 \tag{6.8}$$

Now we have pinned down the specific session that the test query will be asked, and because of the previous games we also know that during that session the adversary is an outsider and passive with respect to all parties.

Game 5: In this game we change the way *Send* queries are answered during the test session, so that the adversary has no way to distinguish between this and the previous game without breaking the CDH assumption. Here is how the simulator works in this game:

In round 1 of the test session, all messages y_i are chosen at random from $\mathbb{G}$ instead of being discrete logarithm public keys. In round 2, all t_i^R are assigned random values from $\{0,1\}^k$, instead of being hashed Burmester-Desmedt secrets. All other computations are performed as in Game 4.

The only way that an adversary can distinguish between this and the previous game, would be to calculate the discrete logarithm x_i of y_i, and query the random oracle for $H(y_{i+1}^{x_i})$. Then the adversary could check that the random oracle output $H(y_{i+1}^{x_i})$ did not match the t_i^R message of the participant. Let *Ask* be such an event:

$$|Pr[S_5] - Pr[S_4]| \leq Pr[Ask]$$

and the advantage of the adversary on this game is:

$$\begin{aligned}\tau_4 &= |2Pr[S_4] - 1| = \\ &\leq |2Pr[S_4] - 2Pr[S_5]| + |2Pr[S_5] - 1| \\ &\leq 2Pr[Ask] + \tau_5\end{aligned} \tag{6.9}$$

If the event *Ask* occurs as part of $\mathcal{A}_{ake}$, we can use $\mathcal{A}_{ake}$ to solve the CDH problem in $\mathbb{G}$. That

is, we can construct a CDH solver Δ that given a CDH instance (g, g^a, g^b) can compute g^{ab}. The construction follows similar logic as the $\mathcal{F}$ construction of Game 1:

During the test session, Δ initializes all participants as normal. During Round 1, Δ picks a participant instance π_U^i and sets $y_i = g^a$ and $y_{i+1} = g^b$. If the event Ask happens in this test session, it means that $\mathcal{A}_{ake}$ might have queried the random oracle for $H(g^{ab})$.

The adversary, in her attempt to distinguish this game, could have picked any (y_i, y_{i+1}) pair of public keys to break. There are n such pairs, hence Δ has at least $1/n$ chance that the target CDH instance was broken. Furthermore, the probability that a randomly chosen entry from the random oracle table is the target CDH solution is at least $1/q_r$ where q_r is the maximum number of random oracle queries allowed. This gives us:

$$Succ^{cdh} \geq \frac{1}{nq_r} Pr[Ask]$$

which can be rewritten as

$$Pr[Ask] \leq nq_r Succ^{cdh}$$

and since $Succ^{cdh}$ is negligible by definition, the event Ask occurs with negligible probability only.

Game 6: This game is the same as the previous game except that in the test session the game halts if $\mathcal{A}_{ake}$ asks a random oracle query with input $(pid_i \| k_1 \| \dots \| k_n)$.

Because of the previous game, random values were XORed with the key shares k_i, and hence the protocol messages in round 2 carry no information about $k_1 \dots k_n$. The best that an adversary can do in this case is to guess their values and since they are n k-bit strings, the probability that the adversary will guess correctly is $(1/2^k)^n$. Finally, the probability that $\mathcal{A}_{ake}$ will ask the right random oracle query during the test session is at most $q_r/2^{kn}$. Which gives us:

$$|Pr[S_6] - Pr[S_5]| \leq \frac{q_r}{2^{kn}}$$

$$\begin{aligned} \tau_5 &= |2Pr[S_5] - 1| \\ &\leq |2Pr[S_5] - 2Pr[S_6]| + |2Pr[S_6] - 1| \\ &\leq 2\frac{q_r}{2^{kn}} + \tau_6 \end{aligned} \tag{6.10}$$

However, if the adversary is not able to query the random oracle H on the correct input then the adversary has no advantage in distinguishing the real session key from a random one and

so:

$$\tau_6 = 0 \tag{6.11}$$

By combining equations 6.2 to 6.11 we calculate the final advantage to be as claimed:

$$\begin{aligned} Adv_{\mathcal{A}_{ake}} &\leq 2Pr[Forge] + 2Pr[Collision] + 2Pr[Repeat] + 2Pr[Ask] + 2\frac{q_r}{2^{kn}}) \\ &\leq 2\left(n^2 Adv^{cma} + \frac{((n+2)q_s + q_r)^2}{2^k} + \frac{q_s^2}{2^k} + nq_r q_s Succ^{cdh} + \frac{q_s q_r}{2^{kn}}\right) \end{aligned}$$

□

Chapter 7

Conclusions and Future Directions

While the scientific field of secure messaging is young, in recent years we see increased interest by researchers and engineers to understand the problem space and design usable solutions.

Since secure messaging systems have so many different applications, we need better understanding of their use cases and the properties they require. For example, are we talking about 3 friends chatting, or a 2000 people group? How should additional people join the chat (with invitation, password, ...)? Do we want people to see when others are online or not (presence-hiding)? Do we expect clients on shaky unreliable networks or not (mobile phones)? By better understanding these requirements we also comprehend the tradeoffs that are involved which allow us to design useful secure messaging systems. Of course, robust usability studies are also required to finally create usable tools that will have maximal impact.

We believe it's important to develop such tools as free software, and also design them in the open so that researchers can be involved easily. In recent years, we are happy to see discussion lists designed specifically for secure messaging discussions [1] [2].

On the cryptographic side, by better understanding the tradeoffs of secure messaging we can design group key exchanges optimized for this specific use case. In the future, it's important to look into extended security properties that could be used in such an application (anonymous, unlinkable, deniable key exchange). We should also optimize the computational costs of our key exchange protocols, to be able to facilitate groups of large size. Finally, the development of robust security models for group key exchange is very important as is further research on the categorization and taxonomy of such security models and the threats they handle.

[1] https://moderncrypto.org/

[2] https://lists.cypherpunks.ca/mailman/listinfo/otr-dev

Appendix A

Mathematical Background

A.1 Computational Assumptions

We now describe some computational assumptions which form the basis of security for the GKE protocols that we describe in the later chapters.

Definition 4 (Discrete Logarithm Assumption)**.** *Let $\mathbb{G}$ be a cyclic group of prime order p and let g be an arbitrary primitive root of $\mathbb{G}$. The discrete logarithm of $g^a \in \mathbb{G}$ to the base g is the unique integer $a \in \mathbb{Z}_p$. The discrete logarithm problem (DLP) is computing the discrete logarithm of g^a to the base g, given a random instance (g, g^a). We say that the discrete logarithm assumption holds in $\mathbb{G}$ if for all Probabilistic Polynomial-Time (PPT) algorithms, the probability of solving the DLP in $\mathbb{G}$ is negligible in a given security parameter k.*

Definition 5 (Computational Diffie-Hellman (CDH) Assumption)**.** *Let $\mathbb{G}$, g, p be as described above. The CDH problem is to compute g^{ab} given a random instance (g, g^a, g^b) for $a, b \in \mathbb{Z}_p$. We say that the CDH assumptions hold in $\mathbb{G}$ if for all PPT algorithms, the probability of solving the CDH problem is negligible in a given security parameter k. Note that the hardness of the CDH problem entails that of the DLP.*

A.2 Crypto Background

Negligible Function

Informally, we call a function f *negligible* if it approaches zero faster than the reciprocal of any polynomial. More concretely:

Definition 6 (Negligible Function)**.** *A function $f : \mathbb{N} \to \mathbb{R}$ is* negligible *if for every positive*

polynomial p there exists an $N \in \mathbb{N}$ such that for all $n < N$,

$$f(n) < \frac{1}{p(n)}$$

To illustrate the power of this definition, consider an adversary who wants to guess a k-bit string. Since there are 2^k possible strings of size k, an attacker who tries a random string has probability $1/2^k$ of guessing it right. 2^{-k} is a negligible function (following the definition above), which means that even if the attacker does a constant number of attempts or even a number of attempts polynomial in k, the attacker would still have a negligible probability of guessing the actual string.

A.2.1 Digital Signatures

A digital signature is a cryptographic scheme for demonstrating the authenticity of a digital message. It's an asymmetric cryptography construction, hence it uses private and public keys. Informally, given a message, Alice can generate a valid signature for that message using her private key. The verifier Bob, given a message and its signature, can verify that the message was signed by the owner of a specific private key. A digital signature scheme typically consists of three algorithms:

- A *key generation algorithm* that selects a private key uniformly at random from a set of possible private keys. The algorithm outputs the private key and a corresponding public key.
- A *signing algorithm* that, given a message and a private key, produces a signature.
- A *signature verifying algorithm* that given a message, public key and a signature, either accepts or rejects the message's claim to authenticity.

To better demonstrate the power of this construction, we now describe the notion of *UF-CMA security* for digital signature schemes. Informally, *UF-CMA security* for signature schemes requires that an adversary cannot produce a signature on a message that has not been signed before.

Unforgeability under Chosen-Message-Attack (UF-CMA security)

Definition 7 (UF-CMA security)**.** *A digital signature scheme is* UF-CMA secure *if the advantage of any PPT adversary in the following game is negligible in the security parameter k.*

The game is carried out in three phases:

Setup The challenger runs the KeyGen algorithm and obtains a key pair (pk, sk). The public key pk is given to the adversary. Also, a public signing oracle is instantiated which can produce signatures of arbitrary messages using sk.

Signing Oracle Queries The adversary can issue queries to the signing oracle with arbitrary input messages $m_1, \ldots, m_q$, for a q polynomial in k. The signing oracle replies with a signature of each message produced using sk. The signing queries may be asked adaptively i.e., the message m_i is allowed to be selected after obtaining the responses to messages $m_1, \ldots, m_{i-1}$.

Forgery Eventually, the adversary outputs a signature (m, σ). The adversary wins the UF-CMA game if (1) σ is valid signature on m under pk and (2) m has not been an input to any of the sign queries i.e. $m \notin \{m_1, \ldots, m_q\}$.

The advantage of an adversary in the UF-CMA game is defined to be the same as its probability of success in the game.

Appendix B

Implementation of a Deniable Group Key Agreement

In this section we provide a research-level implementation of the *Deniable Group Key Agreement* protocol by Bohli et al. [6]. Our implementation is written in Go, and it simply simulates a key exchange session, verifies that it completes successfully and that all participants derive the same key.

```
package main

import (
        "crypto/sha256"
        "math/big"
        "fmt"
        "bytes"
        "container/list"
)

const (
        // security_param is equal to the output length of SHA256
        security_param = 32 // 32 * 8 = 256

        // number of simulated participants
        // feel free to change this, but should be > 2
        n_participants = 8
)

var (
        // group order
        group_q *big.Int
```

```
        // group generator
        group_g *big.Int
        // group prime
        group_p *big.Int
)

// linked list of participants
var participants list.List

// Round 1:
// Where each participant commits to her key share, and publicizes her
// circular group key agreement public key.
func (p *Participant) round_1() {
        var hashBytes [security_param]byte

        // Compute phase
        // k <- {0,1}^k
        p.k = rand_bytes(hashBytes[:])
        // x <- Z_q
        p.x = rand_int(group_q)
        // y = g^x
        p.y = new(big.Int).Exp(group_g, p.x, group_p)

        // Broadcast phase
        // Compute commitment
        h := sha256.New()
        h.Write(p.k)
        h.Sum(p.h_k[:0])
        // Broadcast it
        fmt.Printf("[M-1-%d BROADCAST START]\n", p.i)
        fmt.Printf("H(k_%d) = %x\ny_%d = %d\n", p.i, p.h_k, p.i, p.y)
        fmt.Printf("[M-1-%d BROADCAST END]\n", p.i)

        // Print debugging info
        debug.Printf("Participant %d:\nk = 0x%x\nx = %d\ny = %d\nh_k = 0x%x",
                p.i, p.k, p.x, p.y, p.h_k)
}

// Round 2:
// Where each participant generates a session id, and publicizes her
// Schnorr short-term public key
func (p *Participant) round_2() {
        // Compute phase
        // sid = H(pid || H(k_1) || H(k_2)|| ... || H(k_n))
        h := sha256.New()
```

```
        h.Write(p.pid)
        for p_tmp := participants.Front(); p_tmp != nil; p_tmp = p_tmp.Next() {
                h.Write(p_tmp.Value.(*Participant).h_k[:])
        }
        h.Sum(p.sid[:0])

        // r <- Z_q
        p.r = rand_int(group_q)
        // z = g^r
        p.z = new(big.Int).Exp(group_g, p.r, group_p)

        debug.Printf("Participant %d:\nr = 0x%x\nz = 0x%x\n",
                p.i, p.r, p.z)

        // Broadcast phase
        fmt.Printf("[M-2-%d BROADCAST START]\n", p.i)
        fmt.Printf("sid_%d = 0x%x\nz_%d = 0x%x\n", p.i, p.sid, p.i, p.z)
        fmt.Printf("[M-2-%d BROADCAST END]\n", p.i)
}

// Round 3:
// Where each participant calculates the shared-secret for her
// neighbours and then publicizes her encrypted commited value.
func (p *Participant) round_3() {
        // Compute phase
        prev_p := p.get_prev_participant_circular().Value.(*Participant)
        next_p := p.get_next_participant_circular().Value.(*Participant)
        debug.Printf("%d. previous: %d. next: %d", p.i, prev_p.i, next_p.i)

        // t_L = H(y_{i-1}^x)
        h_l := sha256.New()
        to_hash_tmp_l := new(big.Int).Exp(prev_p.y, p.x, group_p)
        h_l.Write(to_hash_tmp_l.Bytes())
        h_l.Sum(p.t_l[:0])
        debug.Printf("prev_p.y = 0x%x\np.x = 0x%x\nto_hash_tmp_l = 0x%x\nt_L = 0x%x",
            prev_p.y, p.x, to_hash_tmp_l, p.t_l)

        // t_R = H(y_{i+1}^x)
        h_r := sha256.New()
        to_hash_tmp_r := new(big.Int).Exp(next_p.y, p.x, group_p)
        h_r.Write(to_hash_tmp_r.Bytes())
        h_r.Sum(p.t_r[:0])
        debug.Printf("next_p.y = 0x%x\np.x = 0x%x\nto_hash_tmp_r = 0x%xt_L = 0x%x", next_p
            .y, p.x, to_hash_tmp_r, p.t_r)
```

```
        // t = t_L ^ t_R
        xored_tmp := xor_bytes(p.t_l[:], p.t_r[:])
        copy(p.t[:], xored_tmp)
        debug.Printf("t_L = 0x%x\nt_R = 0x%x\nt = 0x%x", p.t_l, p.t_r, p.t)

        // Broadcast phase
        // broadcast (k ^ t_R, t, U_i)
        encrypted_commited_value := xor_bytes(p.k, p.t_r[:])
        copy(p.encrypted_commited_value[:], encrypted_commited_value)

        fmt.Printf("[M-3-%d BROADCAST START]\n", p.i)
        fmt.Printf("encrypted_commit = 0x%x, t = 0x%x\n", p.encrypted_commited_value, p.t)
        fmt.Printf("[M-3-%d BROADCAST END]\n", p.i)
}

// Verify that the round1 commitment of participant 'i' was legit.
// Return true if the commitment was legit; otherwise false.
func (p *Participant) commitment_was_legit(i int) (bool) {
        // If clockwise, to get k_i from the PoV of participant 'i-n':
        //      k_i ^ t_i^R ^ T_i ^ T_{i-1} ^ T_{i-2} ^ T_{i-n+1} ^ H(y_{i-n+1}^x{i-n})
        // Example:
        // to get k_5  from the PoV of 2:
        //      k_5 ^ t_5^R ^ T_5 ^ T_4 ^ T_3 ^ H(y_3^x_2)
        // Let's begin by finding participant 'i'

        // Get target participant
        p_target := get_participant_element(i)
        if p.i == p_target.Value.(*Participant).i {
                debug.Printf("Asked to verify our own commited value (k_%d): Trivially
                    true.", p.i)
                return true
        }

        debug.Printf("[-] Computing k_%d from the PoV of %d:", p_target.Value.(*
            Participant).i, p.i)

        xor_list := make([][]byte, 0) // list that contains all the values that must be
            xored
        // add the encrypted k_d to the XOR list
        debug.Printf("Appending k_%d XOR t_%d^R: 0x%x.",
                p_target.Value.(*Participant).i, p_target.Value.(*Participant).i, p_target
                    .Value.(*Participant).t[:])
        xor_list = append(xor_list, p_target.Value.(*Participant).encrypted_commited_value
            [:])
        p_tmp := p_target
```

```
        // Loop participants in a clockwise fashion
        for {
                if p_tmp.Value.(*Participant).i == p.i {
                        // we got from participant 'i' to 'p'.
                        // time to stop.
                        debug.Printf("Reached %d. Bailing.", p_tmp.Value.(*Participant).i)
                        break
                }

                if p_tmp.Value.(*Participant).i == ((p.i + 1) % n_participants ) {
                        // append H(y_i^x_j)
                        dh_shared_secret := new(big.Int).Exp(p_tmp.Value.(*Participant).y,
                             p.x, group_p)
                        h := sha256.New()
                        h.Write(dh_shared_secret.Bytes())
                        hashed_dh := h.Sum(nil)
                        xor_list = append(xor_list, hashed_dh[:])
                        debug.Printf("Appending H(y_%d^x_%d): 0x%x.", p_tmp.Value.(*
                             Participant).i, p.i, hashed_dh[:])
                }

                // add T_j to the list
                xor_list = append(xor_list, p_tmp.Value.(*Participant).t[:])
                debug.Printf("Appending T_%d: 0x%x.", p_tmp.Value.(*Participant).i, p_tmp.
                     Value.(*Participant).t[:])

                p_tmp = get_prev_circular(p_tmp)
        }

        // xor everything together
        xor_panic := xor_bytes(xor_list[0], xor_list[1:]...)

        if bytes.Equal(xor_panic, p_target.Value.(*Participant).k) {
                // commitment was legit
                debug.Printf("Found legit commitment k_%d: 0x%x", p_target.Value.(*
                     Participant).i, xor_panic)
                return true
        }

        fmt.Printf("xor_panic k_%d: 0x%x", p_target.Value.(*Participant).i, xor_panic)
        fmt.Printf("for realz k_%d: 0x%x", p_target.Value.(*Participant).i, p_target.Value
             .(*Participant).k)
        return false
}
```

```
// Do the verify phase of round 4.
func (p *Participant) round4_verify() (bool) {
        // Verify that t_1 ^ t_2 ^ t_3 ^ ... ^ t_n = 0
        pre_xor := participants.Front().Value.(*Participant).t
        for p_tmp := participants.Front().Next() ; p_tmp != nil ; p_tmp = p_tmp.Next() {
                xor_step := xor_bytes(pre_xor[:], p_tmp.Value.(*Participant).t[:])
                copy(pre_xor[:], xor_step)
        }
        debug.Printf("[-] VERIFY that pre_xor is zeroes: pre_xor = 0x%x", pre_xor)
        // check that the final xor value is 0
        for i := range pre_xor {
                if pre_xor[i] != 0 {
                        fmt.Println("My god! pre_xor: 0x%x", pre_xor)
                        return false
                }
        }

        // for each participant, decrypt her k_j value (ciphertext
        // broadcasted in round3) and check that her round1 commitment
        // was legit:
        for i := 0; i < n_participants; i++ {
                result := p.commitment_was_legit(i)
                if !result {
                        return false
                }
        }

        return true
}

// Round 4:
// Where each participant generates her sessionkey and session
// confirmation. Then they send a Schnorr signature to authenticate
// the hash of their session key and session confirmation.
func (p *Participant) round_4() {
        // Verify phase
        if !p.round4_verify() {
                panic("omg round4_verify failz")
        }

        // Compute phase
        // compute session key: sk_i = H(pid_i || k_i || ... || k_n)
        h_sk := sha256.New()
        h_sk.Write(p.pid)
```

```
        for p_tmp := participants.Front(); p_tmp != nil; p_tmp = p_tmp.Next() {
                h_sk.Write(p_tmp.Value.(*Participant).k[:])
        }
        h_sk.Sum(p.ss[:0])

        // compute session confirmation: sconf_i = H((y_i, k_i) || ... || (y_n, k_n))
        h_sconf := sha256.New()
        for p_tmp := participants.Front(); p_tmp != nil; p_tmp = p_tmp.Next() {
                h_sconf.Write(p_tmp.Value.(*Participant).y.Bytes())
                h_sconf.Write(p_tmp.Value.(*Participant).k[:])
        }
        h_sconf.Sum(p.sconf[:0])

        // compute c_i = H(sid_i || sconf_i) mod q
        h_ci := sha256.New()
        h_ci.Write(p.sid[:0])
        h_ci.Write(p.sconf[:0])
        p.c = new(big.Int).SetBytes(h_ci.Sum(nil))
        p.c.Mod(p.c, group_q)

        // d_i = r_i - c_i * a_i (mod q)
        tmp := new(big.Int).Mul(p.c, p.sk)
        tmp.Mod(tmp, group_q) // is this necessary?
        tmp.Sub(p.r, tmp)
        tmp.Mod(tmp, group_q)
        if tmp.Sign() < 0 {
                tmp.Add(tmp, group_q)
        }
        p.d = tmp

        debug.Printf("sk_i = 0x%x\nsconf_i = 0x%x\nc_i = 0x%x\nd_i = 0x%x\n",
                p.ss, p.sconf, p.c.Bytes(), p.d.Bytes())

        // Broadcast phase
        // broadcast (d_i, U_i)
        fmt.Printf("[M4-%d BROADCAST START]\n", p.i)
        fmt.Printf("d_i = 0x%x\n", p.d.Bytes())
        fmt.Printf("[M4-%d BROADCAST END]\n", p.i)
}

// Final verification:
// Where each participant verifies the Schnorr signatures of the other
// participants.
func (p *Participant) final_verification() (bool) {
        // foreach participant we need to verify that:
```

```
        // g^d_j * PK_j^c_i == z_j (mod p)
        for p_tmp_element := participants.Front() ; p_tmp_element != nil ; p_tmp_element =
             p_tmp_element.Next() {
                p_tmp := p_tmp_element.Value.(*Participant)

                if p_tmp.i == p.i {
                        debug.Printf("Trivial final verification of %d", p.i)
                        continue
                }

                // build g^d_j
                left_hand_side := new(big.Int).Exp(group_g, p_tmp.d, group_p)
                // build PK_j ^ c_i
                left_hand_side_tmp := new(big.Int).Exp(p_tmp.pk, p.c, group_p)
                // build g^d_j * PK_j^c_i
                left_hand_side.Mul(left_hand_side, left_hand_side_tmp)
                left_hand_side.Mod(left_hand_side, group_p)

                right_hand_side := p_tmp.z

                if left_hand_side.Cmp(right_hand_side) != 0 {
                        fmt.Printf("Final verification failure!\nleft_hand_side: 0x%x\
                            nright_hand_side: 0x%x\n",
                                left_hand_side, right_hand_side)
                        return false
                }

                debug.Printf("Final verification (%d verifies %d) (g^d_%d*PK_%d^c_%d == z_
                    %d): 0x%x\n",
                        p.i, p_tmp.i, p_tmp.i, p_tmp.i, p.i, p_tmp.i, right_hand_side)
        }

        return true
}

func main() {
        fmt.Println("Bohli Deniable (M<phase>-<participant>)")

        fmt.Println("[*] Initialization")

        // initialize simulation participants and put them in the linked
        // list
        for i := 0; i < n_participants; i++ {
                participants.PushBack(NewParticipant(i))
        }
```

```
        fmt.Println("[*] Round 1")

        // do round 1
        for p := participants.Front(); p != nil; p = p.Next() {
                p.Value.(*Participant).round_1()
        }

        fmt.Println("[*] Round 2")

        // do round 2
        for p := participants.Front(); p != nil; p = p.Next() {
                p.Value.(*Participant).round_2()
        }

        fmt.Println("[*] Round 3")

        // do round 3
        for p := participants.Front(); p != nil; p = p.Next() {
                p.Value.(*Participant).round_3()
        }

        fmt.Println("[*] Round 4")

        // do round 4
        for p := participants.Front(); p != nil; p = p.Next() {
                p.Value.(*Participant).round_4()
        }

        fmt.Println("[*] Final verification")

        // do final verification
        for p := participants.Front(); p != nil; p = p.Next() {
                if (!p.Value.(*Participant).final_verification()) {
                        panic("final verification failed!")
                }
        }

        // Print results
        p := participants.Front().Value.(*Participant)
        fmt.Printf("[*] Final verification was correct! Finished!\n")
        fmt.Printf("[*] session_key = 0x%x\n", p.ss)
        fmt.Printf("[*] session_confirmation = 0x%x\n", p.sconf)
}
```

```
// initiate crypto parameters (called on startup)
func init() {
        // 2.1.  1024-bit MODP Group with 160-bit Prime Order Subgroup
        // from RFC5114

        // group prime
        group_p, _ = new(big.Int).SetString("
            B10B8F96A080E01DDE92DE5EAE5D54EC52C99FBCFB06A3C69A6A9DCA52D23B616073E28675A23D189838EF1E2EE652C013
            ", 16)
        // subgroup size
        group_q, _ = new(big.Int).SetString("F518AA8781A8DF278ABA4E7D64B7CB9D49462353",
            16)
        // generator
        group_g, _ = new(big.Int).SetString("
            A4D1CBD5C3FD34126765A442EFB99905F8104DD258AC507FD6406CFF14266D31266FEA1E5C41564B777E690F5504F21316
            ", 16)
}
```

Bibliography

[1] Michel Abdalla, Céline Chevalier, Mark Manulis, and David Pointcheval. Flexible group key exchange with on-demand computation of subgroup keys. In Daniel J. Bernstein and Tanja Lange, editors, *AFRICACRYPT 10: 3rd International Conference on Cryptology in Africa*, volume 6055 of *Lecture Notes in Computer Science*, pages 351–368, Stellenbosch, South Africa, May 3–6, 2010. Springer, Berlin, Germany.

[2] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. Cryptology ePrint Archive, Report 2000/014, 2000. http://eprint.iacr.org/2000/014.

[3] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *Advances in Cryptology – CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 232–249, Santa Barbara, CA, USA, August 22–26, 1994. Springer, Berlin, Germany.

[4] Mihir Bellare and Phillip Rogaway. Provably secure session key distribution: The three party case. In *27th Annual ACM Symposium on Theory of Computing*, pages 57–66, Las Vegas, Nevada, USA, May 29 – June 1, 1995. ACM Press.

[5] Ray Bird, Inder S. Gopal, Amir Herzberg, Philippe A. Janson, Shay Kutten, Refik Molva, and Moti Yung. Systematic design of two-party authentication protocols. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 44–61, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Berlin, Germany.

[6] Jens-Matthias Bohli and Rainer Steinwandt. Deniable group key agreement. In Phong Q. Nguyen, editor, *Progress in Cryptology - VIETCRYPT 06: 1st International Conference on Cryptology in Vietnam*, volume 4341 of *Lecture Notes in Computer Science*, pages 298–311, Hanoi, Vietnam, September 25–28, 2006. Springer, Berlin, Germany.

[7] Jens-Matthias Bohli, Maria Isabel Gonzalez Vasco, and Rainer Steinwandt. Secure group key establishment revisited. *Int. J. Inf. Sec.*, 6(4):243–254, 2007.

[8] Nikita Borisov, Ian Goldberg, and Eric Brewer. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, pages 77–84, New York, NY, USA, 2004. ACM.

[9] E. Bresson, O. Chevassut, D. Pointcheval, and J.-J. Quisquater. Provably authenticated group diffie-hellman key exchange. page 255–264. ACM Press, November 2001.

[10] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. Dynamic group Diffie-Hellman key exchange under standard assumptions. In Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 321–336, Amsterdam, The Netherlands, April 28 – May 2, 2002. Springer, Berlin, Germany.

[11] Emmanuel Bresson and Mark Manulis. Securing group key exchange against strong corruptions. In *Proceedings of the 2008 ACM symposium on Information, computer and communications security*, pages 249–260. ACM, 2008.

[12] Mike Burmester and Yvo Desmedt. A secure and efficient conference key distribution system (extended abstract). In Alfredo De Santis, editor, *Advances in Cryptology – EUROCRYPT'94*, volume 950 of *Lecture Notes in Computer Science*, pages 275–286, Perugia, Italy, May 9–12, 1995. Springer, Berlin, Germany.

[13] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. *J. ACM*, 51(4):557–594, 2004.

[14] Qingfeng Cheng and Chuangui Ma. Security weakness of flexible group key exchange with on-demand computation of subgroup keys. *CoRR*, abs/1008.1221, 2010.

[15] Kim-Kwang Raymond Choo, Colin Boyd, and Yvonne Hitchcock. Errors in computational complexity proofs for protocols. In Bimal K. Roy, editor, *Advances in Cryptology – ASIACRYPT 2005*, volume 3788 of *Lecture Notes in Computer Science*, pages 624–643, Chennai, India, December 4–8, 2005. Springer, Berlin, Germany.

[16] George Danezis. Should group key agreement be symmetric and contributory?, 2014.

[17] Whitfield Diffie, Paul C. van Oorschot, and Michael J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, June 1992.

[18] eQualit.ie. (n+1)sec.

[19] Tilman Frosch, Christian Mainka, Christoph Bader, Florian Bergsma, Jörg Schwenk, and Thorsten Holz. How secure is textsecure? *IACR Cryptology ePrint Archive*, 2014:904, 2014.

[20] Ian Goldberg, Berkant Ustaoğlu, Matthew Van Gundy, and Hao Chen. Multi-party Off-the-Record Messaging. In *Proceedings of the 16th ACM Conference on Computer and Communications Security (CCS)*, pages 358–368, New York, NY, USA, November 2009. ACM Press.

[21] M. Choudary Gorantla, Colin Boyd, and Juan Manuel González Nieto. Modeling key compromise impersonation attacks on group key exchange protocols. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009: 12th International Conference on Theory and Practice of Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 105–123, Irvine, CA, USA, March 18–20, 2009. Springer, Berlin, Germany.

[22] Matthew Van Gundy. Improved Deniable Signature Key Exchange for MpOTR. April 2013.

[23] Jonathan Katz and Moti Yung. Scalable protocols for authenticated group key exchange. *J. Cryptology*, 20(1):85–113, 2007.

[24] Meng Hui Lim, Sanggon Lee, and HoonJae Lee. Insider impersonation attack on a tripartite id-based authenticated key agreement protocol with bilinear pairings. In *Next Generation Web Services Practices, 2008. NWESP '08. 4th International Conference on*, pages 123–128, Oct 2008.

[25] Hong Liu, Eugene Y. Vasserman, and Nicholas Hopper. Improved group off-the-record messaging. In *Proceedings of the 12th ACM Workshop on Workshop on Privacy in the Electronic Society*, WPES '13, pages 249–254, New York, NY, USA, 2013. ACM.

[26] Mark Manulis. Security-Focused Survey on Group Key Exchange Protocols. Technical Report 2006/03, Horst-Görtz Institute, Network and Data Security Group, November 2006.

[27] Moxie Marlinspike. Private group messaging, 2014.

[28] C.J. Mitchell, M. Ward, and P. Wilson. Key control in key agreement protocols. *Electronics Letters*, 34(10):980–981, May 1998.

[29] Kyungah Shim. Cryptanalysis of al-riyami-paterson's authenticated three party key agreement protocols. Cryptology ePrint Archive, Report 2003/122, 2003. http://eprint.iacr.org/2003/122.

[30] Victor Shoup. On formal models for secure key exchange. Technical Report RZ 3120, IBM, 1999.

[31] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith. Sok: Secure Messaging. *IEEE Security & Privacy 2015*, 2015.