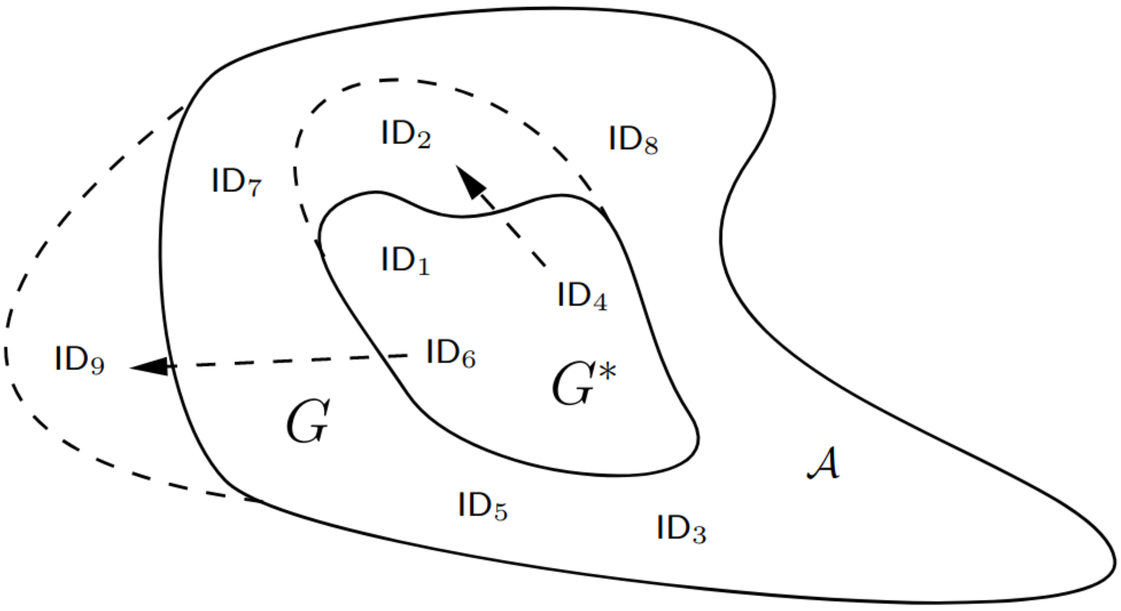
DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2021

On Secure Administrators for Group Messaging Protocols

DAVID BALBÁS GUTIÉRREZ



**KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE**

On Secure Administrators for Group Messaging Protocols

David Balbás Gutiérrez

A thesis submitted in partial fulfilment of the requirements for the degree of

Master of Science in Computer Science

KTH Royal Institute of Technology
School of Electrical Engineering and Computer Science

Swiss Federal Institute of Technology Lausanne
Laboratory of Security and Cryptography (LASEC)

August 2021

Supervisor (EPFL):	Prof. Serge Vaudenay
Advisor (EPFL):	Daniel P. Collins
Supervisor (KTH):	Dr. Douglas Wikström
Examiner (KTH):	Prof. Johan Håstad

Abstract

In the smartphone era, instant messaging is fully embedded in our daily life. Messaging protocols must preserve the confidentiality and authenticity of sent messages both in two-party conversations and in group chats, in which the list of group members may suffer modifications over time. Hence, a precise characterization of their security is required.

In this thesis, we analyze the cryptographic properties that are desirable in secure messaging protocols, particularly in asynchronous group key agreement protocols. Our main contribution is a study of the administration of a messaging group, which is a common scenario in which a subset of the group members (the *administrators*) are the only users allowed to modify the group structure by adding and removing group members. As we discuss, enabling secure group administration mechanisms can enhance the security of messaging protocols.

For this purpose, we introduce a new primitive which extends the *continuous group key agreement* (CGKA) primitive to capture secure administration, which we denote by *administrated CGKA* (A-CGKA). The definition is followed by a correctness notion and an informal security description. We present two constructions of our A-CGKA that can be built on top of any CGKA: *individual admin signatures* (IAS), and *dynamic group signature* (DGS), both constructed using signature schemes.

Furthermore, we provide a detailed overview of secure group messaging in which we discuss group evolution, efficiency, concurrency, and different adversarial models. We introduce a novel CGKA correctness definition (in the so-called *propose-and-commit* paradigm), followed by a security game that incorporates the correctness properties. We also survey some variants of the *TreeKEM* protocol and compare their security.

Key words: Secure Messaging – Group Messaging – Ratcheting – Group Key Agreement – Cryptographic Administrators – Group Administration

Sammanfattning

I de smarta telefonernas tid är direktmeddelanden en självklar del av vår vardag. Meddelandeprotokoll måste upprätthålla konfidentialitet och autenticitet för skickade meddelanden både i tvåpartskonversationer samt i gruppchatter vars medlemslistor kan förändras över tid. Därför krävs en precis karaktärisering av deras säkerhet.

I detta arbete analyserar vi de kryptografiska egenskaper som är önskvärda i meddelandeprotokoll med fokus på asynkrona gruppnyckelavtalsprotokoll (group key agreement protocols). Arbetets huvudsakliga bidrag till området är en studie av administrationen av en meddelandegrupp. Detta är ett vanligt förekommande scenario där endast en delmängd av gruppmedlemmarna (*administratörerna*) tillåts modifiera gruppens struktur genom att lägga till och ta bort medlemmar. Som diskuteras i arbetet kan användandet av säkra gruppadministrationsmekanismer (group administration mechanisms) förbättra säkerheten för meddelandeprotokoll.

I detta syfte introducerar vi en ny kryptografisk primitiv vilken uttökar den s.k. "*continuous group key agreement*"-primitiven (CGKA) till att även innefatta säker administration. Denna primitiv kallar vi *administrated CGKA* (A-CGKA), vars definition följs av en korrekthetsdefinition och en informell säkerhetsbeskrivning. Vi presenterar två konstruktioner av A-CGKA som kan byggas ovanpå vilken CGKA som helst: *individual admin signatures* (IAS) och *dynamic group signature* (DGS), som båda konstrueras via signaturscheman.

Utöver detta ger vi även en detaljerad överblick över säkra gruppmeddelanden i vilken vi diskuterar gruppevolution, effektivitet, samtidighet och olika fientliga modeller. Vi introducerar en ny definition av korrekthet för CGKA (vilket följer paradigmen *propose-and-commit*) följt av ett s.k. "security game" som inkorporerar korrekthetsegenskaperna. Vi undersöker även varianter av *TreeKEM*-protokollet och jämför deras säkerhet.

Acknowledgments

This Master Thesis is the outcome of six fantastic months of work at the Laboratory of Security and Cryptography (LASEC), at EPFL in Lausanne. I am grateful to Serge Vaudenay for hosting me in the lab, for supervising the project, and for teaching me the importance of rigour and formalism in cryptography. I would also like to thank everyone at LASEC for welcoming me at the lab and for creating such a nice and supportive atmosphere.

A special mention goes to my advisor, colleague, and friend Daniel Collins. Our uncountable hours of discussions have been priceless for this project. Thanks so much for your patience and dedication. I learned a lot from you - this project does not finish here, and I hope that our collaboration endures in time.

This work also concludes two great years of studies at KTH. I am grateful to Johan Håstad for accepting the role as examiner, but also for guidance and teaching during my time in Stockholm.

I would also like to thank Phillip Gajland for proofreading, macros, and insightful comments; and Filip Hörnsten for his help with the "sammanfattning". Last but not least, this project (and my studies in general) would not have been possible without my friends and family - thanks so much for your unconditional support.

Contents

List of Figures

Chapter 1

Introduction

Billions of people use instant messaging services daily. Building messaging protocols that provide privacy and authenticity is a nontrivial task for many reasons. One of them is that users must be able to exchange messages *asynchronously*; these can be sent or received at any time, without previous agreement, and without users being online at all times. Besides, sessions are long-lived in contrast to protocols such as TLS, and the secrets are stored in potentially vulnerable devices such as smartphones, which have a large attack surface. Hence, instant messaging protocols should provide security in particularly challenging scenarios.

Secure messaging protocols are divided in two large families: two-party messaging and group messaging. Two-party messaging considers the scenario in which two parties Alice and Bob communicate securely. Extending two-party messaging protocols to groups, in which more than two group members exchange messages, presents additional challenges.

1.1 Secure Messaging

Following the literature (see for example [ACD19, ACDT20, DV19, JMM19]), there are four basic desirable properties of a secure messaging protocol.

- **Correctness:** Assuming that all parties honestly follow the protocol, all messages sent by a user should be received without modification by the intended receivers.
- **Authenticity:** Users should be able to ensure that received messages come from their claimed sender.
- **Confidentiality:** An adversary should not be able to learn any information about the content of sent messages (except possibly their length).

- **Asynchronicity:** Users are not be required to be online at all times; correctness, authenticity, and confidentiality must be preserved if they are offline.

These features are commonly required in most secure communication protocols, and can be achieved with standard cryptographic tools given that parties remain secure (their keys are not leaked, their randomness sources are not manipulated, their devices are not hacked, etc.). As we discussed, it is likely that this is not always the case in messaging. Hence, the following additional properties are also desirable:

- **Forward Security (FS):** If one of the parties is compromised or if a key is exposed, this should not affect the confidentiality of previous messages.
- **Post-Compromise Security (PCS):** Confidentiality should be restored after a temporary key leakage. In other words, after a state compromise, the communication should self-heal following the exchange of a few messages [CGCG16].

Forward Security. This property can be achieved using symmetric cryptography. A simple solution is for parties to hash communication keys regularly, as we now illustrate for the two-party case. Assume that Alice (A) and Bob (B) share a key k_0 that they use to send messages to each other, using a symmetric encryption scheme. Given a key derivation function KDF (for example HMAC), they can obtain a new key $\mathsf{k}_1 = \mathsf{KDF}(\mathsf{k}_0, t_1)$, where t_1 is some arbitrary input such as a predefined constant[1]. Because of the security provided by the KDF, generally built upon secure hash functions, an adversary given k_1 will not be able to read messages encrypted under k_0. This process is seen in Figure 1.1.

$$\begin{tikzcd}
 & t_1 \arrow[d] & t_2 \arrow[d] & t_3 \arrow[d] & \\
 & \mathsf{KDF} \arrow[d] & \mathsf{KDF} \arrow[d] & \mathsf{KDF} \arrow[d] & \cdots \\
\mathsf{k}_0 \arrow[ur] & \mathsf{k}_1 \arrow[ur] & \mathsf{k}_2 \arrow[ur] & \mathsf{k}_3 \arrow[ur] &
\end{tikzcd}$$

Figure 1.1: Simple key ratchet for forward security.

Post-Compromise Security. This concept was first formally introduced in [CGCG16], although the idea was nascent in protocols such as Off-The-Record (OTR) [BGB04] and Signal [PM16]. Assume that A and B share a symmetric key k_0 as before, and suppose that an adversary is able to learn the state of A, including k_0,

[1]For example, the original implementation of the Signal protocol [Mar] uses `0x01` and `0x02` as predefined constants for the derivation of chain keys and message keys.

at a given point in time. If the adversary is passive (i.e., it does not try to interfere in the communication, for example by impersonating A), A and B can carry out a key-exchange protocol such as Diffie-Hellman to obtain a new symmetric key. Since it is hard to determine in practice when a compromise occurs, a solution is to include fresh key-exchange material every time a message is sent.

This approach is relatively expensive, since it requires public-key cryptography. However, it is very useful, as we can see with two attack scenarios. First, assume that an eavesdropper (i.e., a passive attacker) is recording the communications between A and B. If the attacker manages to obtain one of the keys in the future, he will only be able to read a few messages; as soon as the communication self-heals, there will be a new key that cannot be derived from previous ones. Second, assume that an active attacker is trying to impersonate one of the parties. In a PCS scenario, the attacker will need to get access to the keys immediately. If there is any delay, communication will have self-healed and the key will have evolved. Besides, if the attacker compromises one of the users but the user manages to recover, security is fully re-established too.

Additional notions. PCS and FS are not the only security goals considered in the literature. Protocols may pursue different notions of *anonymity* or *deniability* [BGB04, HW20]. In addition, protocols may or may not rely on a central server that provides message ordering, which is particularly relevant in group messaging [ACDT20, WKHB20]. On the other hand, the assumptions on the network may greatly differ; some protocols are proven secure against active adversaries that can drop and inject messages [ACJM20] or schedule messages arbitrarily [KPPW$^+$21], whereas other offer protections only against passive adversaries [ACDT20]. The diversity of security notions and assumptions is reflected in the literature, which includes many constructions built upon different definitions.

1.2 Motivation

In group messaging, most of the difficulties from two-party messaging remain (such as asynchronicity), while new challenges arise. One of them is *efficiency*: simple approaches such as running a two-party messaging protocol between every pair of users scale poorly with the group size. Another challenge concerns group *evolution* or *dynamics*: the list of group members may change at any point in time, requiring complex key agreement protocols. To provide sound security properties, formal proofs are essential and require the specification of an adversarial model.

In some of the most popular messaging apps in the world, such as WhatsApp [Wha16], Signal Messenger [Mar], or Telegram [Tel], group dynamics are managed by a subset of group members, called *group administrators*. These are the only members allowed to perform specific actions such as adding and removing group participants. As we argue in Chapter 4, including secure group administration mechanisms may positively affect the security of the group by mitigating insider attacks [AJM20, KS05] and implementation faults [RMS18], reducing concurrency issues [BDR20], and limiting the trust on a central infrastructure. Nevertheless, group administration is currently not enforced using cryptography, but only at the application level (and possibly with the help of central servers). To the best of our knowledge, this problem has not been formally studied in the literature before.

1.3 Goals

The main research question of this thesis is: *how can we build efficient and provably secure group messaging protocols?*

This question is very broad; hence, we focus on three specific problems.

1. What are the mechanisms that enable forward security and post-compromise security in messaging protocols, and how can we characterize their precise security?
2. What are the main challenges in the construction of group messaging and group key agreement protocols, and what adversarial models are considered?
3. How can we construct cryptographic administrators for group messaging, and what security properties can we achieve with them?

The first problem is mostly addressed in Chapters 2 and 3 via a literature review which examines existing protocols. The second problem is addressed in Chapter 3, where we review the literature but also present our own analysis and definitions. The third problem is addressed in Chapter 4, where all material is novel work.

1.4 Scope

Secure messaging can be studied under different perspectives. One example is privacy, where one may consider for example anonymity and metadata protection. Another example is distributed computing, where parties can be faulty or even adversarial. Our focus is on provable security - we are interested in protocols whose security can be formally based on standard cryptographic assumptions. The results that we present in this thesis are limited, since they are a snapshot of an ongoing research project. Therefore, we will likely introduce substantial changes to the current constructions, including progress on some of the future work directions that we mention. Among these, proving the security of our constructions is among our highest priorities.

More broadly, many of the techniques that we study in this thesis, such as ratcheting, key exchange, and protection against active adversaries, have applications in many other fields. Hence, the scope of our analysis is not limited to secure messaging. Besides, we believe that our idea of group administrators may be extended to cryptographic protocols of diverse flavours, particularly when changes in the set of participating instances require specific security measures.

1.5 Outline

In this thesis, we address secure group messaging, and in particular asynchronous group key exchange protocols, from a cryptographic perspective. In Chapter 2, we provide an overview of two-party messaging and ratcheting, introduce formal definitions and develop intuition for the core concepts. In Chapter 3, we introduce group messaging, group key agreement (and in particular, the notion of *continuous group key agreement* (CGKA) [ACDT20]), and the different adversarial models. Noting the lack of suitable correctness notions in the literature, we present a novel notion of CGKA correctness which focuses on dynamic membership, and a detailed discussion of the properties that should be achieved. We also provide a detailed overview of the TreeKEM protocol and some of its variants [BBR18].

In Chapter 4, we dive into the problem of group administration and introduce the *administrated continuous group key agreement* (A-CGKA) primitive, a correctness notion, and some security considerations. We present two different A-CGKA constructions built from CGKAs and signature schemes, that we denote by *individual admin signatures* (IAS) and *dynamic group signature* (DGS), and discuss their properties and further possibilities for group administrators. In Chapter 5, we conclude the thesis and suggest some lines of future work.

An exposition of the basic cryptographic primitives, assumptions and notation used throughout the thesis can be found in Appendix A. We note that all its content is standard in the literature.

1.6 Contributions

The main scientific contributions of this thesis are the following.

- In Chapter 3, we discuss the desirable properties of group key agreement protocols, leading to our notion of CGKA correctness. Furthermore, we generalize the CGKA syntax by introducing policies and multiple groups.
- In Chapter 4, we motivate how cryptographic administrators enhance the security of group messaging and introduce the A-CGKA primitive. A-CGKA is an extension of CGKA which supports group administration. We also provide a game-based correctness definition and a discussion towards a security definition.

- Following our A-CGKA primitive, we introduce two constructions of group administrators that use signature schemes. Both protocols are modular, practical, and can be implemented on top of any CGKA.

Chapter 2

Two-Party Messaging

Within secure messaging, a large number of protocols address two-party communication. One of the most popular solutions is the Signal protocol [PM16], which inherits many ideas from the OTR protocol [BGB04]. In Signal, users perform a continuous key exchange in which every message is encrypted with a different key to improve security. Following Signal, many protocols have been proposed in the literature in recent years [DV19, ACD19, JS18, PR18], differing in their efficiency and their level of security.

In this chapter, we first introduce the Signal protocol [PM16], which illustrates many of the main ideas in secure messaging, many of them applicable to group messaging. Signal and their predecessors inspired the study of forward and post-compromise secure key exchange, known as ratcheting, that we introduce later. Finally, we survey the existing ratcheting constructions and compare the different security notions.

2.1 Terminology and Notation

We start by introducing several terms that appear frequently in the secure messaging literature.

- A *user*, *participant*, or *party* is an entity that takes part in a messaging (or key exchange) protocol. Users are identified by a unique identity string ID, which is a public parameter.
- The *state* γ of a user ID is the information used for protocol execution. This includes keys, message records, dictionaries, parameters, and in some cases used randomness.
- An *adversary* $\mathcal{A}$ is an idealized external entity which can interact with the protocol via a security game (see Appendix A). The capabilities of an

adversary are often defined by set of *oracles.* These allow the adversary to interact with the protocol by, for example, scheduling message delivery, injecting messages, or leaking parties' states.

- A user ID suffers a *state compromise* or a *corruption* if their state is leaked to the adversary. One possible scenario is that the user's device is remotely controlled by the adversary (for instance stolen or hacked).
- An *impersonation* of a user ID occurs when the adversary forges a message from ID and sends it to other user.
- A *key exposure* occurs when a user ID leaks a single key to the adversary, such as a group key or a symmetric key, without revealing their whole state. This scenario can be seen as a partial corruption.
- A participant is *out-of-sync* if their state does not match the state(s) of the other participant(s). For example, if a participant receives a successfully forged message, or if they remain offline for a long time.
- A *randomness exposure* occurs when the randomness used by a participant in some part of the protocol (such as key generation) leaks. The leakage may occur before or after the algorithm is executed. A *randomness manipulation* occurs when an adversary is able to choose the random values used by the participant.

Game notation. The definitions, games, and constructions that we introduce in our work use standard notation. Algorithms, oracle names, and cryptographic parameters are denoted in sans-serif font. To assign the output of an algorithm Alg on input x to a variable a, we write $a \leftarrow \mathsf{Alg}(x)$ if the algorithm is deterministic. If the algorithm is randomized, we write $a \leftarrow\$ \mathsf{Alg}(x)$. Whenever we want to make the randomness explicit, we write $a \leftarrow \mathsf{Alg}(x; r)$, meaning that r is the randomness used by Alg. An algorithm can input or return blank values, represented by $\perp$. The security parameter is denoted by λ (some basic examples where this notation is used can be found in Appendix A).

In this thesis, we introduce several games played between a challenger and an adversary $\mathcal{A}$ that has access to several oracles. In oracles and algorithms, some special predicates are used. The predicate '**require** P' enforces that a logical condition P is satisfied; otherwise the oracle/algorithm finishes immediately and returns $\perp$. The predicate '**reward** P' is executed in games and is such that if P holds, the adversary immediately wins the game. More precisely, the game returns 1 in games where the advantage is defined by the probability that the adversary outputs 1 (unpredictability), and the game reveals the bit $b \in \{0, 1\}$ to the adversary in indistinguishability games. For logical conditions X and Y, the predicate '**reward** X **iff** Y' is short-hand for '**reward** $(X \wedge Y) \vee (\neg X \wedge \neg Y)$'. The keyword '**public** var' indicates that the adversary has read access to the variable var.

To store and retrieve values, we will often use dictionaries: $\mathsf{A}[k] \leftarrow a$ adds the value a to the dictionary A under key k (overwriting the previous value associated to k). $b \leftarrow \mathsf{A}[k]$ retrieves $\mathsf{A}[k]$ and assigns it to variable b. A dictionary A is initialized as $\mathsf{A}[\cdot] \leftarrow a$; where all values of the dictionary are set to a (notice that a can also be $\perp$).

Initial key exchange and continuous key agreement. Key agreement in a two-party messaging protocol has two components. The first is the initial key agreement, which addresses the problem of two (or more) parties that want to establish a common secret key for starting a communication session. The second component is the *evolution* of such key with the aim of preserving confidentiality during the session. This part is responsible for evolving the keys and providing FS and PCS. We will commonly refer to it as *ratcheting* or *continuous key agreement*. Both are complementary and should be combined in a secure messaging protocol such as Signal.

2.2 The Signal Protocol

Variants of the Signal protocol [PM16] can be found at the core of messaging apps used by billions of people on a daily basis, such as WhatsApp, Signal App, and Facebook Messenger's secret conversations. Signal is an elegant and efficient way to achieve FS and PCS while being highly practical.

2.2.1 Description

Signal builds upon two mechanisms: the Extended Triple Diffie-Hellman (X3DH) protocol and the Double Ratchet Algorithm. The first one performs an initial handshake between two parties, such that they can agree on a shared key. This exchange is mediated by a server which allows Signal users to retrieve public information from other users. Once two parties have agreed on a shared key, the Double Ratchet algorithm deals with ensuring FS and PCS for the (potentially very long-lived) messaging session.

We briefly provide some intuition for the X3DH algorithm. If Alice is a user of the protocol, she publishes a series of Diffie-Hellman public keys and signatures: an identity key IK_A, a signed prekey spk_A, the prekey signature $\sigma = \mathsf{Sig}(\mathsf{IK}_A, \mathsf{spk}_A)$ where spk_A is signed under IK_A, and several one-time prekeys $\{\mathsf{OPK}_{A_1}, \ldots, \mathsf{OPK}_{A_r}\}$ that are provided for asynchronicity. Then, if Bob wants to communicate with Alice, she will perform a series of DH key agreements between the keys from Alice and his own keys (and vice versa). The derived keys are then hashed into a single key k_{AB}. In total, four simultaneous DH key agreements are performed to derive k_{AB}. Two are performed to authenticate each user and the other two provide forward security.

If a X3DH session is completed successfully, Alice and Bob will have a common *root key* to initiate the Double Ratchet Algorithm. As the name suggests, the algorithm introduces two synchronized ratchets: a symmetric-key ratchet, and a Diffie-Hellman ratchet. The symmetric ratchet follows a similar approach as Figure 1.1, providing FS. The users keep track of a KDF chain (which is derived from the initial root key) and derive further chain keys via a KDF. Every time the KDF is applied on the previous chain key, it refreshes it and generates a message key, used to encrypt (or decrypt) the messages sent. The process is shown in Figure 2.1.

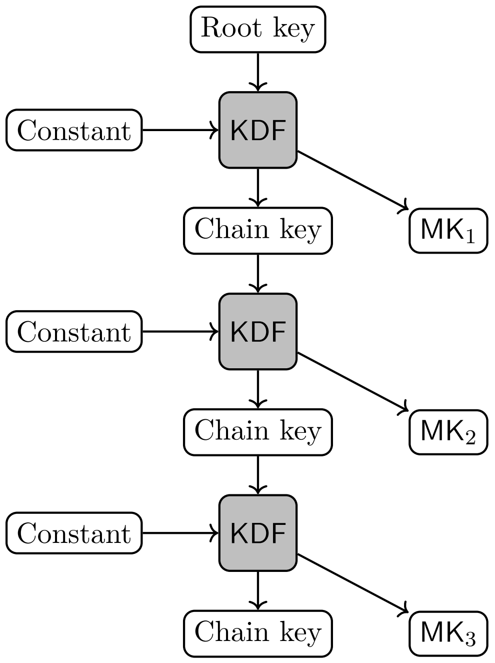


Figure 2.1: Scheme of a KDF chain that outputs forward secure message keys MK_i (based on [PM16, Gaj20]).

Each user keeps two simultaneous KDF chains: one for received messages (the receiver chain), and one for sent messages (the sender chain). Every time they send or receive a message, they obtain the corresponding message key by updating their chains. Since they both start at the same root key and the key derivation is deterministic, they will be synchronized.

A subtle aspect is the behaviour of the algorithm when a message is delivered out-of-order. Suppose for example that Alice sends three messages m_1, m_2, m_3. Alice will then update the sender chain three times, obtaining three message keys $\mathsf{MK}_1, \mathsf{MK}_2, \mathsf{MK}_3$ and producing corresponding ciphertexts $c_i = \mathsf{Enc}(\mathsf{MK}_i, M_i)$, that she will send in order. Now, suppose that Bob receives first c_3, then c_1 and finally c_2. Upon reception of c_3, Bob will advance its receiver chain and try to decrypt c_3 using MK_1, which will fail. To fix this problem, Signal introduces a parameter in the message metadata that indicates the message number. Bob then will know that the received ciphertext c_3 corresponds to the third message. Therefore, he will advance the chain three times while storing MK_1 and MK_2, and use MK_3 to decrypt c_3. Later, when Bob receives c_1, c_2, it will use the stored message keys. This useful feature is called *immediate decryption* and implies a small sacrifice on forward security. Namely, if an adversary compromises Bob while he is waiting for c_1, c_2, she will obtain MK_1 and MK_2.

The DH ratchet introduces new randomness every time the sender/receiver roles switch during a conversation. Whenever this happens, we say that an *epoch* has passed. For example, if during epoch n Alice is sending messages to Bob, then Bob replies (epoch $n+1$) and Alice replies again, the parties will be in epoch $n+2$. The shared DH secrets are used to update the root key, i.e., the key that is used to derive the chain keys and message keys of the symmetric ratchets. Hence, message keys that belong to different epochs cannot be derived from each other. After two epochs (in which the adversary must remain passive), the conversation self-heals from a potential state compromise.

The DH ratchet works as follows. Suppose Alice starts a new epoch as a sender. First, she generates a DH key pair $\mathsf{sk}_A = a, \mathsf{pk}_A = g^a$, and sends pk to Bob with her first message. When Bob advances the epoch and sends a message to Alice, he will generate $\mathsf{sk}_B = b, \mathsf{pk}_B = g^b$ and calculate $I_0 = g^{ab}$, which is the DH shared key. Bob will update the root key of his sending chain using I_0, calculate chain and message keys, and send his message to Alice using these new keys, which he attaches pk_B to. Once Alice receives pk_B, she will calculate g^{ab}, update her receiving chain, and read Bob's message. The key-exchange is carried out continuously; when Alice sends a new message, she creates a pair $\mathsf{pk}'_A = g^{a'}, \mathsf{sk}'_A = a'$, updates her sender chain with $g^{a'b}$ and sends pk'_A to Bob, etc. The full mechanism is seen in Figure 2.2.

2.2.2 Abstraction

The security of Signal was first formally studied by Cohn-Gordon et. al. [CGCD+20]. They studied both the X3DH algorithm and the Double Ratchet. Later, Alwen et. al. studied the Double Ratchet in detail [ACD19] and suggested a modularization of the protocol in three components, as follows.

- The *forward-secure authenticated encryption with associated data (FS-AEAD)* models the symmetric key ratchet of Signal, in which message keys are derived, and the encryption of the application messages.

- The *continuous key agreement (CKA)* models the DH ratchet of Signal and can be built from any public-key encryption scheme. This abstraction can be extended to group messaging, as we will see in Chapter 3. A nice feature is that the CKA admits the replacement of the DH protocol with any key encapsulation mechanism (KEM), including post-quantum secure KEMs [ACD19].

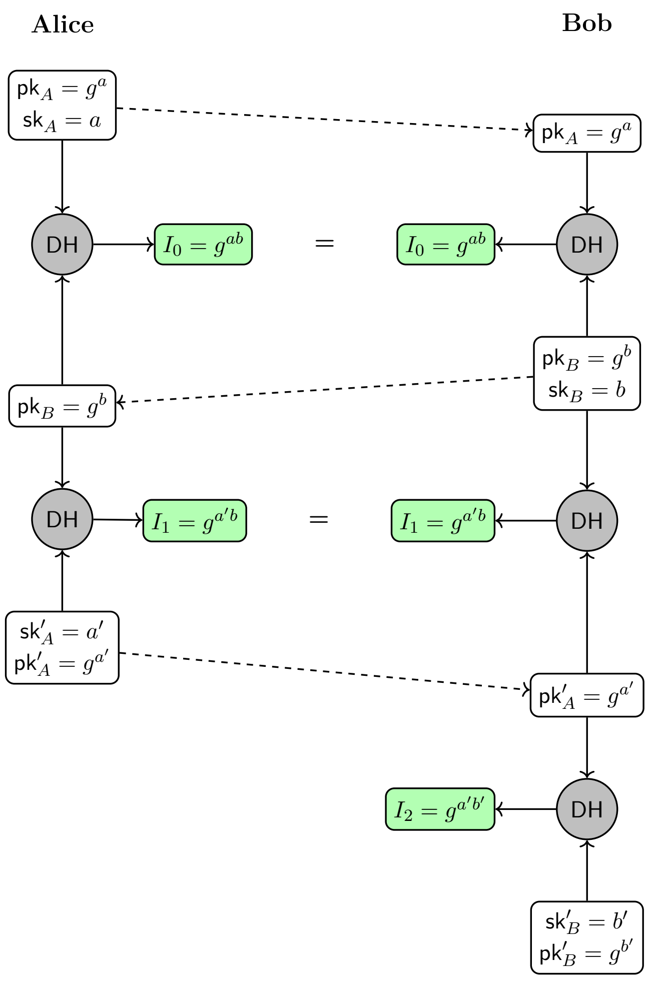


Figure 2.2: Diagram of the Double Ratchet Algorithm (based on [PM16, Gaj20]). The green nodes represent the shared keys, which are then included in the key schedule of the sending and receiving chains.

- The *PRF-PRNG* is a double-input hash function that connects both ratchets, introducing the shared randomness obtained by the CKA in the key schedule of the FS-AEAD.

Despite its great usability, practicality and elegance, one can argue that Signal offers insufficient security in certain scenarios. If an adversary compromises the full state of Alice, then during the current epoch, both the confidentiality and the authenticity of future messages sent by Alice are lost. Besides, since the states are symmetric, Bob loses all security too. The scenario in which Alice loses authenticity and Bob loses confidentiality seems unavoidable. Namely, the adversary can impersonate Alice and read the messages that Bob sends to Alice, since it has Alice's state. However, as pointed out in [ACD19], there is no reason to believe that Alice should not be able to send future messages that remain private to the adversary (if she has access to a good source of randomness). Similarly, it should be possible that Bob's messages remain authenticated. Achieving such goals, however, comes at the price of less efficient and more complex protocol constructions [PR18, JS18].

2.3 Ratcheting as a Primitive

Both the OTR protocol [BGB04] and Signal's Double Ratchet protocol were highly influential in the design of later ratcheted key exchange protocols, which present different security and efficiency properties. Bellare et. al. [BSJ+17], were the first to study ratcheting (also ratcheted key exchange or asynchronous ratcheted key agreement) as a standalone cryptographic primitive. Their work left many open directions, and it was continued by several others; first [PR18, JS18], followed by [JMM19, DV19, CDV21].

2.3.1 Definition

The work by Bellare et. al. considered only the unidirectional case, i.e., when Alice has a fixed sender role and Bob has a fixed receiver role. In their model, only the sender can be compromised, and the receiver cannot send messages. Such a protocol may be useful in some situations, but is clearly unsatisfactory for messaging. The first constructions of full, bidirectional asynchronous ratcheting key agreement protocols (BARK in this work, although it appears under multiple names in the literature) were introduced by Poettering and Rösler [PR18] and Jaeger and Stepanovs [JS18]. We introduce the formal definition of a BARK following [DV19].

Definition 2.1 (BARK)**.** *A bidirectional asynchronous ratcheted key agreement is a tuple of algorithms* $\mathsf{BARK} = (\mathsf{setup}, \mathsf{gen}, \mathsf{init}, \mathsf{send}, \mathsf{receive})$ *such that:*

- $\mathsf{pp} \leftarrow\!\$\, \mathsf{setup}(1^\lambda)$ *determines common public parameters* pp*.*
- $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\$\, \mathsf{gen}(1^\lambda, \mathsf{pp})$ *generates a private-public key pair* $(\mathsf{sk}, \mathsf{pk})$ *for a participant.*
- $\mathsf{st}_{\mathsf{ID}} \leftarrow \mathsf{init}(1^\lambda, \mathsf{pp}, \mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}'}, \mathsf{ID})$ *sets the initial state* $\mathsf{st}_{\mathsf{ID}}$ *of the participant* $\mathsf{ID} \in \{A, B\}$ *given its secret key and the public key of the other participant* $\mathsf{ID}' \neq \mathsf{ID}$*.*
- $(\mathsf{st}'_{\mathsf{ID}}, \mathsf{upd}, \mathsf{k}) \leftarrow\!\$\, \mathsf{send}(\mathsf{st}_{\mathsf{ID}})$ *outputs an updated state* $\mathsf{st}'_{\mathsf{ID}}$ *for the sender* ID'*, an update message* upd*, and a key* k*.*
- $(\mathsf{acc}, \mathsf{st}'_{\mathsf{ID}}, \mathsf{k}) \leftarrow \mathsf{receive}(\mathsf{st}_{\mathsf{ID}}, \mathsf{upd})$ *outputs a bit* $\mathsf{acc} \in \{\mathsf{true}, \mathsf{false}\}$ *to indicate acceptance or rejection of* upd*, an updated state* $\mathsf{st}'_{\mathsf{ID}}$*, and a key* k*.*

We remark on two features of the above definition. First, public-key cryptography is introduced, and one can in fact prove that a BARK which offers PCS requires a KEM [DV19]. Second, the notion of ratcheting is captured by the output key k of the send and receive algorithms. The protocol is run asynchronously between the two participants A and B, meaning that they can initiate a send operation at any time without prior agreement.

2.3.2 Correctness

To ensure the correctness of a BARK, we require that the receiver keys for A are generated in the same order as the sender keys by B and vice-versa. In other words, a BARK is correct if both parties generate matching keys after honest delivery.

The correctness of a BARK can be formalized by an unpredictability game $\mathsf{CORR}_{\mathsf{BARK}}$ as in [DV19][1]. The game starts by calling an initialization oracle $\mathcal{O}^{\mathsf{Init}}$ that sets up the states of both parties. Then, the adversary can arbitrarily interact with two oracles $\mathcal{O}^{\mathsf{Send}}$ and $\mathcal{O}^{\mathsf{Receive}}$ that run the protocol. $\mathsf{CORR}_{\mathsf{BARK}}$ will always return 0 unless a **reward** clause is satisfied; in this case the algorithm returns 1. The events that are rewarded are: that the keys generated by A and B do not match, and that the reception of an honestly delivered message (i.e., a call to $\mathsf{receive}$) fails and returns $\mathsf{acc} = \mathsf{false}$.

Definition 2.2 (Correctness of BARK). *A BARK is correct if, for all λ and all computationally unbounded adversaries $\mathcal{A}$, it holds that:*

$$\Pr[\mathsf{CORR}^{\mathcal{A}}_{\mathsf{BARK}}(1^{\lambda}) = 1] = 0,$$

where the probability is taken over the choice of the challenger and adversary's random coins.

We note that the correctness considers computationally unbounded adversaries, meaning that there exists no sequence of messages between both parties that makes the protocol fail if the messages are delivered in order. Besides, a well-formed correctness game should ensure that trivial protocols that might satisfy security notions (such as a trivial key exchange protocol that always outputs $\perp$) do not satisfy Definition 2.3.

2.3.3 Security

There exist a variety of security notions for a BARK. Depending on the actual instantiation of the protocol, one can define games that capture different levels of security, differing in notions such as security against randomness exposure or manipulation, the number of exchanged messages required for state recovery, etc. As a rule of thumb, the stronger the security definition, the heavier the protocol needs to be.

A general framework to capture the security of a BARK is using a key indistinguishability game [DV19, ACD19, JS18, PR18], that leads to so-called KIND

[1] In Chapters 3 and 4, we will introduce detailed games for group messaging (and in particular, for the Continuous Group Key Agreement primitive).

security. A $\mathsf{KIND}^{\mathcal{A}}_{b,\mathsf{C}_{\mathsf{clean}}}(1^\lambda)$ game is parametrized by a security parameter λ, a bit $b \in \{0, 1\}$, and a predicate $\mathsf{C}_{\mathsf{clean}}$. After the state of both parties A and B has been initialized, an adversary $\mathcal{A}$ can interact with a variety of oracles that control the protocol. At any point, $\mathcal{A}$ may make a party send or receive a message (using the send and $\mathsf{receive}$ algorithms). Depending on the security model, $\mathcal{A}$ may also drop messages, alter the message ordering, inject forged messages, expose the state of a participant, expose or manipulate randomness, etc. Finally, the adversary will be able to *challenge* any of the participants at a given point by calling a challenge oracle.

A challenge to a participant ID is carried out similarly as in an IND-CPA game: $\mathcal{A}$ produces two plaintexts m_0, m_1 of the same length and a participant sends the ciphertext corresponding to m_b to the other party, using the usual send algorithm. At some point, $\mathcal{A}$ must stop the security game and make a guess b'. The cleanness predicate $\mathsf{C}_{\mathsf{clean}}$ is made to exclude trivial attacks against the protocol, such as challenging a participant right after exposing its state or exposing the receiver of a challenge message. This can be either captured in the game itself (as in [PR18]), or be set as a separate predicate (as in [DV19]). The predicate is also used to capture the exact security of the protocol. If a protocol achieves weaker security, the cleanness predicate will exclude a larger number of attacks.

Definition 2.3 (Security of BARK)**.** *A* BARK *is* (λ, q, ϵ)-$\mathsf{C}_{\mathsf{clean}}$*-secure if, for any polynomial-time adversary* $\mathcal{A}$ *limited to* q *oracle queries, the advantage*

$$\mathsf{Adv}^{\mathsf{kind}}_{\mathcal{A}}(\lambda) = \left|\Pr[\mathsf{KIND}^{\mathcal{A}}_{0,\mathsf{C}_{\mathsf{clean}}}(1^\lambda) = 1] - \Pr[\mathsf{KIND}^{\mathcal{A}}_{1,\mathsf{C}_{\mathsf{clean}}}(1^\lambda) = 1]\right|$$

is bounded by ϵ*.*

2.4 Ratcheting Constructions

In this section, we survey different ratcheting protocols and the security properties that they achieve. We note that some of them study ratcheting as a BARK (or any analogous term for asynchronous ratcheted key exchange), i.e., as a key exchange protocol, whereas others construct a messaging scheme where A and B establish a confidential and authenticated communication channel. Both approaches rely on similar properties and constructions but they differ in form. Without loss of generality, we can consider them equivalent since messaging can be built from a BARK and vice versa as stated in [CDV21].

2.4.1 Strong Security

Poettering and Rösler presented the first construction of a full bidirectional ratcheting protocol achieving strong security [PR18]. They introduced the concept of a *key-*

updatable encapsulation mechanism (kuKEM), which is a KEM with the additional feature that private/public key pairs can be updated from old keys.

More precisely, a kuKEM is a tuple of algorithms $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Upd})$ where the first three behave as a KEM and the Upd algorithm has the following functionality. If it receives a secret key sk and update data Δ, it outputs an updated secret key sk'. If it receives a public key pk and Δ, it outputs an updated public key pk'. The correctness of a kuKEM requires that the recipients are synchronized, i.e., that if $(\mathsf{sk}_0, \mathsf{pk}_0) \leftarrow \mathsf{Gen}(1^\lambda)$ is a valid key pair, then for $\Delta_1, \ldots, \Delta_n$, the keys $\mathsf{pk}_i = \mathsf{Upd}(\mathsf{pk}_{i-1}, \Delta_i)$ and $\mathsf{sk}_i = \mathsf{Upd}(\mathsf{sk}_{i-1}, \Delta_i)$ form valid key pairs $(\mathsf{sk}_i, \mathsf{pk}_i)$. The achieved security is that keys encapsulated over $\mathsf{sk}_j, \mathsf{pk}_j$ remain secure even if some key pair $\mathsf{sk}_i, \mathsf{pk}_i$ for $j < i$ is compromised.

In a parallel and independent work, Jaeger and Stepanovs introduce a messaging protocol that provides similar security [JS18]. They achieve security against randomness exposure before usage. Their construction relies on two main cryptographic blocks, following the key-updatable primitive idea: *key-updatable digital signatures (kuDS)*, and *key updatable public-key encryption (UPKE)*. As before, they define an algorithm Upd that takes a public/secret key and an update information Δ and outputs an updated key. Both [JS18] and [PR18] build kuKEM and UPKE from a hierarchical identity-based encryption scheme (HIBE, Definition A.9) [GS02, HL02]. In a HIBE, a secret key sk associated to an identity vector Δ can generate secret keys for identities for which Δ is a prefix, i.e., for identities $\Delta' = (\Delta, \Gamma)$ in a lower hierarchy than Δ. Moreover, any user who has an identity of the form Δ' cannot recover the secret keys for upper hierarchies such as Δ. Namely, to construct a kuKEM/UPKE from a HIBE, it suffices to take the vector of updates Δ as the HIBE's identity vector. With this construction, an adversary that compromises the secret key sk_i associated to Δ_i will not be able to recover any Δ_j for $j < i$.

2.4.2 Practical Security

The main drawback of UPKE's and kuKEM's is precisely that HIBE is a heavy cryptographic primitive to use. Moreover, kuKEM's are in fact necessary to obtain a strongly secure BARK [BRV20]. Thus, they turn out to be impractical in many real-world scenarios. Durak and Vaudenay (DV) introduce a BARK protocol that relaxes the security requirements while improving performance and still achieving better security than Signal [DV19]. Roughly, the main security difference between DV and the above protocols is that in DV, if the state of A is exposed, then no kind of forgery can be received by B (i.e., an update sent by the adversary impersonating A) while B is waiting for a real update. In Jaeger-Stepanovs and Poettering-Rösler, some kind of forgeries are allowed. The DV protocol also introduces strong authentication and unforgeability security notions. This notion captures active adversaries that may attempt to forge ciphertexts sent during a conversation.

Separately, Jost et. al. propose a messaging protocol that achieves an almost-optimal security, in the sense that its security is stronger than in the Durak-Vaudenay protocol but weaker than Jaeger-Stepanovs and Poettering-Rösler [JMM19]. Their construction consists on a slightly weaker variant of UPKE that they call HkuPKE, based on the ElGamal cryptosystem and hence not post-quantum secure.

There is work in the literature that relaxes security further such as [CDV21], that introduces a lighter protocol for ratcheting. New randomness is not introduced every time, but only *on-demand*. In [YV20], this protocol is made more efficient.

2.4.3 Alternative Notions

The correctness of the previous ratcheting constructions depends on a correct order of delivery of the messages. A different approach is taken by Alwen et. al., who look at the immediate decryption property of Signal and model the (often realistic) scenario in which messages may be lost or delayed [ACD19]. They sketch (but leave the security proof for future work) a more secure protocol than Signal, based on standard public-key encryption and signatures.

The immediate decryption scenario leaves many open questions as the protocols achieve weaker security than the above protocols. Intuitively, this is reasonable since when out-of-order messages are allowed, the receiver needs to store the keys for previous messages. A state compromise would then endanger forward security. An interesting line of work would be to characterize what is the best possible security that can be achieved with this feature.

Finally, a notion called *recover* security is introduced both in [DV19] and [JMM19]. The idea, later refined by [CDV21], consists in including a hash of the conversation transcript in every message. In this way, if an adversary compromises A and sends a forged message to B, both parties will notice the forgery as soon as A or B send another message themselves. Namely, if A sends a new message, it will not include the forged message in the transcript, whereas if B sends a message, he will include the forged message and A will notice that the transcripts do not match. This allows both parties to interrupt the communication. The study of recover security with immediate decryption is also of interest, perhaps having direct applicability to the Signal protocol.

Chapter 3

Group Messaging

Most messaging apps such as WhatsApp, Signal and Telegram provide group messaging functionality, in which users communicate with all members of a group simultaneously. With respect to two-party messaging, group messaging presents additional challenges related to efficiency, synchronization, group evolution, and especially security. In this chapter, we introduce group messaging and some important constructions in the literature, putting an emphasis on key agreement, group management, and formal correctness and security notions.

3.1 Overview

Following [UDB+15, Wei19] (with some modification), a group messaging protocol must address four global problems: *communication*, or how messages are delivered (broadcast, pairwise channels, central servers, etc.); *encryption*, or how users agree on their shared encryption keys and ensure confidentiality for their messages; *authentication*, or how users prove authorship (including possible deniability); and *consistency*, or how to handle concurrency and possible different views of the conversation by different users. Depending on how each of the parts is handled, protocols will have different security properties and efficiency. We also identify the problem of *group management*, which is how the group handles membership changes safely. Nowadays, messaging groups are *dynamic*, meaning that they support arbitrary membership changes such as the addition and removal of users.

Multiple keys. A first, natural approach to group messaging is to extend the two-party ratcheting to the group case. This leads to the so-called *pairwise channels* protocol, which runs the Signal Protocol between every pair of users in the group. This protocol is used by Signal Messenger [Mar] on groups with a small number of users. Whenever a group member sends a message, it updates its key ratchet with every other member separately. Unfortunately, the solution scales poorly (quadratically) with the number of users. If a group consists of 20 users, we would need 190 two-party channels. If 20 more members were added to the group, 780 channels are required. Moreover, each time a single user wants to perform a key update to provide PCS, 39 key exchange protocols would need to be run simultaneously.

To reduce the communication complexity from quadratic to linear, WhatsApp [Wha16] uses the so-called *sender keys* approach, where each member encrypts messages using their own (sender) key. These keys are shared with the group and refreshed after every message using a symmetric ratchet. The downside is that PCS is weakened on behalf of efficiency. The keys are refreshed only when a member leaves the group (since otherwise the removed member could continue reading group messages), which involves $\mathcal{O}(n^2)$ operations for a group of n users.

Group key agreement. Most of the solutions in the literature, however, follow a *group key agreement (GKA)* approach where all group members run a protocol to derive a common group key. GKA protocols have been studied for many years. Early work considers static groups in which the participants are fixed, and dynamic groups in which keys cannot be ratcheted [PRSS21]. The scenario in which dynamic groups can *efficiently* update their keys for FS and PCS was addressed only recently by protocols such as Asynchronous Ratchet Trees (ARTs) [CGCG$^+$18] and TreeKEM [BBR18].

Messaging Layer Security. The design of an efficient protocol for group messaging that provides FS and PCS is the aim of the Messaging Layer Security (MLS) [OBR$^+$] workgroup of the Internet Engineering Task Force (IETF). At the core of MLS, we find the *TreeKEM* protocol [BBR18], which is responsible for generating and distributing fresh key material among the members of the group and achieves sub-linear average complexity. These common group secrets are used to derive keys for the encryption of the messages using an authenticated symmetric encryption scheme such as a forward-secure AEAD [ACD19]. The security of the MLS key schedule was recently analyzed in [BCK21].

3.1.1 Delivery Service

Any messaging protocol needs to rely on a service that distributes messages among the group members. We refer to this as the *Delivery Service (DS)*, following MLS [BBM$^+$]. In MLS, the Delivery Service is assumed to provide a reliable, in-order and consistent transcript of messages[1]. This means that messages are delivered in the same order to all users, which is the order of reception by the server, and that every message is eventually delivered.

[1]Besides, the DS of MLS acts as a key directory (also key service), providing users with public key material that they can use to start communicating with other users. Such material must be signed under long-term identity keys provided by the Authentication Service.

The assumptions made by MLS on their DS are rather strong. In short, the MLS protocol requires a central server that routes all messages and that is relatively trusted by all group members. A malicious server could alter the delivery of messages and acknowledgements, provide the users with wrong keys, store data, etc. Therefore, a basic requirement is that confidentiality and authenticity should never be affected by a malicious DS; only protocol correctness. Therefore, the DS is often modelled as adversarial in security proofs, such as in [ACDT20].

3.1.2 Authentication Service

The Authentication Service (AS) provides a mapping between user identities (such as phone numbers or emails) and long-term identity keys. As opposed to the DS, a malicious AS will break confidentiality and authenticity, since it will allow external adversaries to impersonate group members [OBR$^+$]. The amount of trust put into this service is therefore very large.

In some protocols, the AS is modelled as a *public-key infrastructure* (PKI). The PKI is generally trusted and used as a black-box in security proofs [ACDT20, KPPW$^+$21, ACJM20], although it is modelled more realistically when insider adversaries are considered [AJM20]. In order to detect man-in-the-middle attacks or malicious PKIs, WhatsApp, Telegram, and Signal provide optional out-of-band key verification mechanisms.

3.1.3 Concurrency and Decentralization

The central server assumption from MLS prevents that messages are delivered at the same time or in a different order to different parties. This assumption is sometimes unfeasible or undesired [Wei19]. Dealing with concurrent application messages might not be particularly problematic from a usability perspective, but concurrent control messages can totally break security in some protocols [ACDT20].

There exists work towards the decentralization of group messaging protocols. One possible avenue is reaching consensus, as in distributed computing protocols. Nevertheless, this approach is expensive. A different approach was taken in [Wei19] with the Causal TreeKEM protocol, which is a variant of early versions of TreeKEM that supports concurrent group operations. Later, [WKHB20] introduced the decentralized CGKA (DCGKA) abstraction and constructed a DCGKA protocol[2].

To avoid the need of a central server and consensus, [Wei19] and [WKHB20] rely on a causal ordering of messages, meaning that users may have different (but still causally ordered) versions of the conversation transcript. In [WKHB20], a message m_1 causally

[2]The DCGKA name can be misleading, since the protocol in [WKHB20] does not derive a common group key (so that it is not a CKA protocol), but rather implements group messaging through an improved *sender keys* approach.

precedes m_2 (denoted by $m_1 < m_2$) if (i) ID sends message m_1 and then m_2; (ii) ID receives m_1 and then sends m_2; or (iii) there exists m_3 such that $m_1 < m_3$ and $m_3 < m_2$. Both works, however, present relevant limitations and leave many open directions. For instance, their security games do not consider randomness manipulation, assume the existence of a PKI (which is not straightforward in a decentralized setting), and PCS is not ensured in the case of concurrent group operations.

Without dropping the central server assumption, [BDR20] analyzes the cost of achieving concurrent updates among group members. Their work provides a lower bound for the communication complexity of a protocol that supports t concurrent PCS updates.

3.2 Group Key Agreement

Group key agreement (GKA) protocols, sometimes called group key exchange (GKE) in the literature, are a natural approach to group messaging. The family of GKA protocols is very large since there are many possible trade-offs for security, efficiency, and group synchronization (see [BK17] for an efficiency-based and [PRSS21] for a security- and modelling-based comparison of several of them). In this section we provide an overview of GKA, focusing on the continuous group key agreement (CGKA) abstraction, group evolution, correctness, concurrency issues, and security.

3.2.1 Desirable Properties

In the following paragraphs, we discuss the different properties that are desirable in a GKA protocol with application to group messaging.

Group dynamics. A first classification of GKA protocols can be done according to group dynamics. Let $G = \{\mathsf{ID}_1, \ldots, \mathsf{ID}_n\}$ be a list of group members. There exist protocols that only support a fixed list of participants, where G remains static; and protocols that support dynamic changes on G, namely where additions and removals of participants are allowed [PRSS21]. While it is true that one can perform a new fixed-group key exchange every time there is a membership change, dynamic protocols are more efficient and possibly asynchronous (non-interactive). In our work, we focus on GKA protocols that support dynamic membership and asynchronicity.

Protocol syntax. A GKA protocol can be defined by either global algorithms, which manage the overall execution of the protocol, or local algorithms. A local view, in which every participant runs their own instance of the protocol, is preferable in group messaging, partially due to the requirement of asynchronicity. Indeed, most of the recent group messaging works in the literature [ACDT20, KPPW$^+$21, CHK21, BBR18, AJM20, ACJM20] implement local functionality. Nevertheless, none of these considers multiple groups in their protocol definition, even if this is a common feature in practice.

Key partnering. During the execution of the local instances of a protocol, different parties will compute the same (or related) keys under different circumstances (such as a common group key). We refer to this as key partnering. Generally speaking, two instances share a pair of partnered keys whenever the exposure of one of the keys reveals non-trivial information about the other. Constructing a precise definition of partnering is important to show that different group keys are independent, and because of its role in the definition of correctness [PRSS21]. One possibility is using key identifiers; two keys are partnered whenever they share the same key identifier (for instance, a common group key). Another possibility is to use epochs in a similar way to the two-party case [ACDT20, KPPW$^+$21].

Key evolution (ratcheting). Post-compromise security is a fundamental security property in messaging. A suitable GKA protocol should natively support that users refresh both the group key and their local secrets by introducing new randomness. Alwen et. al. [ACDT20] introduce the *continuous group key agreement* (CGKA) primitive to capture ratcheted key agreement in a dynamic group.

Adversarial models. A large diversity of adversarial models is present in the literature. The most general models consider active adversaries who control the network and who can forge and inject messages in the protocol, as in the two-party case. Nevertheless, constructing protocols that offer (at least partial) security against active adversaries is not an easy task [ACJM20]. Some protocols are proven secure with respect to models that assume honest delivery [ACDT20] or semi-honest delivery [KPPW$^+$21]. Other security notions include anonymity and deniability, as in [LVH13] or [HW20]. Further details are provided in Section 3.2.5.

Authentication and multiple groups. As mentioned in the previous section, some protocols require that there exists a public-key infrastructure (PKI) which is trusted to map user identifiers to their keying material [AJM20]. There exists a connection between the PKI and multi-group security, i.e., the security guarantees when the same user belongs to more than one messaging group. Multiple group memberships and multiple devices (such as a party using their phone and laptop for the same account) are very common, but also hard to prove secure [PRSS21]. The reason is that a security issue affecting one group can propagate to others, particularly when the PKI issues the same identity keys in all groups. Multi-group security for MLS, sender keys and pairwise channels is studied in [CHK21], but many other works such as [ACDT20, KPPW$^+$21, ACJM20] ignore this issue.

We also note that a PKI is not the only solution available for authentication. A more general treatment, where no specific authentication mechanism is assumed, is provided in [PRSS21] by using the so-called *authenticators*. Authentication is a crucial part of GKA protocols, especially when active adversaries are considered. In this thesis, authentication is out of scope; our constructions assume that parties are authenticated in some way.

Summary. We conclude that the CGKA primitive captures the fundamental requirements for group messaging (mainly asynchronicity, support for dynamic groups, and key ratcheting). Hence, most of the material in this chapter will consider the CGKA abstraction, which is introduced next.

3.2.2 Continuous Group Key Agreement

We introduce the definition of a CGKA in the so-called *propose and commit* paradigm. This means that users can broadcast different operation proposals in a given group (e.g. adding/removing members), which are eventually collated into a commit message by some group member. The changes are made effective only when the commit message is processed by users [AJM20, ACJM20]. The evolution of a CGKA in time is captured by *epochs*; a group member advances to a different epoch every time they successfully process a commit message, which implies there is a change in the group structure and/or secret from their perspective.

A relevant difference between the definition in [ACDT20] (and other works such as [KPPW+21]) is that our definition considers the multi-group setting, for which we introduce group identifiers of the form gid. According to our definition, a CGKA supports different groups identified by different values gid running concurrently. Group identifiers for CGKAs are introduced in works that use the universal composability (UC) model [ACJM20, AJM20] or consider multi-group security [CHK21].

Definition 3.1. *A continuous group key agreement (CGKA) scheme is a tuple of algorithms* $\mathsf{CGKA} = (\mathsf{init}, \mathsf{create}, \mathsf{prop}, \mathsf{commit}, \mathsf{proc}, \mathsf{key}, \mathsf{policy})$ *such that:*

- $\gamma \leftarrow\$ \mathsf{init}(1^\lambda, \mathsf{ID})$ *takes a security parameter* 1^λ *and an identity* ID *and outputs an initial state* γ*.*

- $(\gamma', T) \leftarrow\$ \mathsf{create}(\gamma, \mathsf{gid}, G)$ *takes a state* γ*, a (unique) group identifier* gid*, and a list of group members* $G = \{\mathsf{ID}_1, \dots, \mathsf{ID}_n\}$ *and outputs a new state* γ' *and a control message* T *(welcome), where* $T = \perp$ *denotes failure.*

- $(\gamma', T) \leftarrow\$ \mathsf{prop}(\gamma, \mathsf{gid}, \mathsf{ID}, \mathsf{type})$ *takes a state, a group identifier, an ID, and a proposal type* $\mathsf{type} \in \mathsf{types} = \{\mathsf{add}, \mathsf{rem}, \mathsf{upd}\}$*, and outputs a new state* γ' *and a proposal message* P*, where* $P = \perp$ *denotes failure. Proposal messages contain the fields* $P.\mathsf{ID}$ *and* $P.\mathsf{type}$*, indicating the* ID *of the user affected by the proposal and the proposal type, respectively.*

- $(\gamma', T) \leftarrow\$ \mathsf{commit}(\gamma, \mathsf{gid}, \vec{P})$ *takes a state, a group identifier, and a vector of proposals* $\vec{P}$*, and outputs a new state* γ' *and a control message* T*, where* $T = \perp$ *denotes failure.*
- $(\gamma', \mathsf{acc}) \leftarrow \mathsf{proc}(\gamma, T)$ *takes a state and a control message* T*, and outputs a new state* γ' *and an acceptance bit* acc*, where* $\mathsf{acc} = \mathsf{false}$ *denotes failure.*
- $k \leftarrow \mathsf{key}(\gamma, \mathsf{gid})$ *takes a state and a group identifier, and outputs the group secret* k *for group* gid*.*
- $G' \leftarrow \mathsf{policy}(\vec{P}, G)$ *takes a vector of proposals and a set of parties* G*, and outputs a set of parties* G'[3]*.*

Finally, given ID*'s state* γ *and* gid*, the (possibly empty) set of group members in* gid *from* ID*'s perspective is stored as* $\gamma[\mathsf{gid}].G$*.*

Protocol execution. Once every user has initialized their state using init, a group is created when a party calls create on a list of IDs. The create algorithm outputs a control message T that must be processed by prospective group members, including the creator, in order to join the group. The create algorithm takes a unique group identifier gid as input; correctness (Figure 3.1) prevents the use of the same gid twice.

Any user can propose an update, add or removal of a group member at any time. This is done via the prop method, which outputs a proposal message P. Proposals encode the information needed to make a change in the group structure or keying material, but they do not immediately make any change in the group. The proposed changes become effective only when a user commits a (possibly empty) vector of proposals $\vec{P} = [P_1, \ldots, P_m]$ using commit. The commit algorithm outputs a control message T that contains the information needed by every group member to process the change, including the users who join the group. The order in which the proposals are committed, as well as other rules (such as allowing proposals from external members or handling contradictory proposals) is specified by the policy algorithm.

Control messages are processed by calling proc, which updates the caller's state and outputs an acceptance bit acc to indicate success or failure. We note that proc does not require a group identifier as input; this models the standard behaviour of a messaging protocol in which, upon reception of a message, the user needs to determine which group the message corresponds to. A concrete description of a protocol (TreeKEM) modelled as a CGKA is presented in Section 3.3.2.

Propose and commit paradigm. In previous group messaging protocols [ACDT20, KPPW⁺21], including old versions of MLS, group changes are made directly by the proposers. Instead of having prop and commit algorithms, specific

[3]Note that this function does not require a state, so it is publicly computable.

'action' algorithms such as $\mathsf{add}, \mathsf{remove}, \mathsf{update}$ made the group changes effective immediately. The *propose and commit paradigm* used in this work was introduced in version 8 of MLS to facilitate that group members refresh their keys frequently [BBM$^+$]. We believe that our approach is more general since protocols such as [ACDT20, KPPW$^+$21] can be written in the propose and commit paradigm - one can simply enforce that the commit algorithm takes a single proposal (or none in case of an update) and must be executed by the proposal creator directly. Besides, this paradigm is more flexible for a group with administrators since one can allow non-admin users to propose group changes if desired, without having any impact on the group until they have been committed (by an administrator).

Overall, in comparison to two-party protocols such as the BARK (Definition 2.1) or to the GKA syntax as in [PRSS21], the CGKA algorithms commit and proc resemble a send and $\mathsf{receive}$ pair. The prop algorithm simply restricts what can be sent.

3.2.3 Correctness

In this section, we aim to build a complete and precise correctness notion for a CGKA. To the best of our knowledge, correctness has not been explicitly defined in the literature. Partial work includes [PRSS21], where a generic correctness game for GKA is provided. In [ACDT20], one of the main correctness properties (the fact that all group members must derive the same group keys) is captured directly in the security game. In the following, we describe desirable CGKA correctness properties, and construct a correctness game that considers them. Such properties can be divided into three families: liveness, consistency, and robustness.

We introduce our notion of CGKA correctness in Definition 3.2, followed by the $\mathsf{CORR}^{\mathcal{A}}_{\mathsf{CGKA}}$ game in Figure 3.1. The correctness game considers an adversary who, through an interaction with a set of oracles, schedules message delivery for honest parties. The honestly generated messages are stored in the dictionary $\mathsf{T}[\cdot]$. The adversary is allowed to schedule different, but consistent sequences of control messages to be processed by the parties involved in a given group. It is not allowed to inject messages; however, it can call the different oracles with malformed inputs (in this case, correct CGKA algorithms must return $\perp$). Besides, the adversary can access the state of each user (stored in $\mathsf{ST}[]$) at all times[4]. Even if randomness manipulation is not mentioned, the adversary is unbounded, so its effects are implicitly considered. We note that the complexity of the CGKA syntax in the propose-and-commit paradigm requires that many checks are performed, resulting in a long (and arguably verbose) game.

[4]Note that the keyword **public** indicates that the adversary has read access to a variable, as defined in Section 2.1.

Definition 3.2 (Correctness of CGKA)**.** *Let* $(\mathsf{OP}, \mathsf{PP}, \mathsf{CP}, \mathsf{DP})$ *denote a tuple of predicates. A* CGKA *is* $(\mathsf{OP}, \mathsf{PP}, \mathsf{CP}, \mathsf{DP})$*-correct if, for all* λ *and all computationally unbounded adversaries* $\mathcal{A}$*, it holds that:*

$$\Pr[\mathsf{CORR}^{\mathcal{A}}_{\mathsf{CGKA}}(1^{\lambda}) = 1] = 0$$

where the probability is taken over the choice of the challenger and adversary's random coins.

Multiple groups. Correctness is defined in the multi-group model. Namely, the adversary may create groups via $\mathcal{O}^{\mathsf{Create}}$ on behalf of different users. Our game ensures that a new group identified by gid can only be created if $\mathsf{create}(\cdot, \mathsf{gid}, \cdot)$ has not been previously called, which is verified via the used dictionary. Group membership is tracked via state variables in the game (stored in the view dictionary), and consistency checks are made with respect to these ($\mathsf{ViewChecker}$ algorithm). We note that our definition allows for users to leave and later rejoin a given group.

Epochs and forking states. Control messages output by successful create and commit calls are uniquely identified. In particular, whenever such a call is made, the corresponding messages are stored in variable T and parametrized by a counter $\mathsf{com\text{-}ctr}$ (similarly for proposals). In a network model where all members receive the same messages in a specific order (such as a central server model), epochs are in one-to-one correspondence with group keys, and therefore define key partnering precisely. However, our model is more general and considers the possibility of forking states, i.e., that there exist partitions of the group in which different group keys are derived. One advantage of this model is that we can apply our correctness property to an active adversarial model in which the adversary can drop messages or prevent their delivery. A second advantage is that we capture concurrent commits that are processed by different group members.

We thus consider epochs as a pair (t, c), where t is an integer which, for a group member, increments each time proc is called (i.e., an epoch index), and c corresponds to the value of $\mathsf{com\text{-}ctr}$ when the message was output. If a user ID is not part of a group gid, we set $\mathsf{ep}[\mathsf{gid}, \mathsf{ID}] = (-1, -1)$. To this end, we can consider parties that, for a given gid and integer t, process different commit messages as long as they provide a consistent view of messages to parties in each partition. For example, consider a group $G = \{A, B, C, D\}$ where both A and C commit at the same time and output messages T_A, T_C respectively. If T_A is processed by A and B, and T_C is processed by C and D, then the group partition $\{A, B\}$ will be in a different epoch than $\{C, D\}$, and the states of the users will be consistent within each partition.

Robustness. The *stateful* algorithms create, commit and proc may fail. In proc, failure is determined by an acceptance bit acc, and in the other two algorithms by

```
$\mathsf{CORR}^{\mathcal{A}}_{\mathsf{CGKA}}(1^\lambda)$
    public $\mathsf{view}[\cdot], \mathsf{ST}[\cdot], \mathsf{T}[\cdot] \leftarrow \bot$
    public $\mathsf{used}[\cdot] \leftarrow \mathsf{false}$   // group id's
    public $\mathsf{prop\text{-}ctr}, \mathsf{com\text{-}ctr} \leftarrow 0$
    public $\mathsf{ep}[\cdot] \leftarrow (-1,-1)$   // not in group
    $\mathcal{A}^{\mathcal{O}}(1^\lambda)$
    return $0$

$\mathcal{O}^{\mathsf{Init}}(1^\lambda, \mathsf{ID})$
    require $\mathsf{ST}[\mathsf{ID}] = \bot$
    $\mathsf{ST}[\mathsf{ID}] \leftarrow\$ \mathsf{init}(1^\lambda, \mathsf{ID})$

$\mathcal{O}^{\mathsf{Prop}}(\mathsf{ID}, \mathsf{gid}, \mathsf{ID}', \mathsf{type})$
    require $\mathsf{ST}[\mathsf{ID}] \neq \bot$
    $(\gamma, P) \leftarrow\$ \mathsf{prop}(\mathsf{ST}[\mathsf{ID}], \mathsf{gid}, \mathsf{ID}', \mathsf{type})$
    reward $P = \bot$ iff $\mathsf{PP}$
    if $P = \bot$ return   // failure
    reward $(P.\mathsf{ID} \neq \mathsf{ID}') \vee (P.\mathsf{type} \neq \mathsf{type})$
    $\mathsf{ViewChecker}(\mathsf{ST}[\mathsf{ID}], \gamma, \mathsf{gid})$
    $\mathsf{prop\text{-}ctr} \leftarrow \mathsf{prop\text{-}ctr} + 1$
    $\mathsf{T}[\mathsf{gid}, \mathsf{ep}[\mathsf{gid}, \mathsf{ID}], \text{`prop'}, \mathsf{prop\text{-}ctr}] \leftarrow P$
    $\mathsf{ST}[\mathsf{ID}] \leftarrow \gamma$

$\mathcal{O}^{\mathsf{Create}}(\mathsf{ID}, \mathsf{gid}, G)$
    require $\mathsf{ST}[\mathsf{ID}] \neq \bot$
    require $\mathsf{used}[\mathsf{gid}] = \mathsf{false}$   // new gid
    $(\gamma, T) \leftarrow\$ \mathsf{create}(\mathsf{ST}[\mathsf{ID}], \mathsf{gid}, G)$
    reward $T = \bot$ iff $\mathsf{OP}$
    if $T = \bot$ return   // failure
    $\mathsf{ViewChecker}(\mathsf{ST}[\mathsf{ID}], \gamma, \mathsf{gid})$
    $\mathsf{used}[\mathsf{gid}] \leftarrow \mathsf{true}$
    $\mathsf{com\text{-}ctr} \leftarrow \mathsf{com\text{-}ctr} + 1$
    $\mathsf{T}[\mathsf{gid}, (-1,-1), \text{`com'}, \mathsf{com\text{-}ctr}] \leftarrow T$
    $\mathsf{ST}[\mathsf{ID}] \leftarrow \gamma$

$\mathsf{ViewChecker}(\gamma_1, \gamma_2, \mathsf{gid})$
    reward $\mathsf{key}(\gamma_1, \mathsf{gid}) \neq \mathsf{key}(\gamma_2, \mathsf{gid})$
    reward $\gamma_1[\mathsf{gid}].G \neq \gamma_2[\mathsf{gid}].G$

$\mathcal{O}^{\mathsf{Commit}}(\mathsf{ID}, \mathsf{gid}, (i_1, \ldots, i_k))$
    require $\mathsf{ST}[\mathsf{ID}] \neq \bot$
    require $\mathsf{ep}[\mathsf{gid}, \mathsf{ID}] \neq (-1,-1)$
    $\vec{P} \leftarrow (T[\mathsf{gid}, \mathsf{ep}[\mathsf{gid}, \mathsf{ID}], \text{`prop'}, i])_{i=(i_1,\ldots,i_k)}$
    $(\gamma, T) \leftarrow\$ \mathsf{commit}(\mathsf{ST}[\mathsf{ID}], \mathsf{gid}, \vec{P})$
    reward $T = \bot$ iff $\mathsf{CP}$
    if $T = \bot$   return   // failure
    reward $\mathsf{ID} \notin \mathsf{ST}[\mathsf{ID}][\mathsf{gid}].G$
    $\mathsf{ViewChecker}(\mathsf{ST}[\mathsf{ID}], \gamma, \mathsf{gid})$
    $\mathsf{com\text{-}ctr} \leftarrow \mathsf{com\text{-}ctr} + 1$
    $\mathsf{T}[\mathsf{gid}, \mathsf{ep}[\mathsf{gid}, \mathsf{ID}], \text{`com'}, \mathsf{com\text{-}ctr}] \leftarrow T$
    $\mathsf{T}[\mathsf{gid}, \mathsf{ep}[\mathsf{gid}, \mathsf{ID}], \text{`vec'}, \mathsf{com\text{-}ctr}] \leftarrow \vec{P}$
    $\mathsf{ST}[\mathsf{ID}] \leftarrow \gamma$

$\mathcal{O}^{\mathsf{Deliver}}(\mathsf{ID}, \mathsf{gid}, (t, c), c')$
    require $\mathsf{ST}[\mathsf{ID}] \neq \bot$
    require $\mathsf{ep}[\mathsf{gid}, \mathsf{ID}] \in \{(t, c), (-1,-1)\}$
    $T \leftarrow \mathsf{T}[\mathsf{gid}, (t, c), \text{`com'}, c']$
    require $T \neq \bot$
    $(\gamma, \mathsf{acc}) \leftarrow \mathsf{proc}(\mathsf{ST}[\mathsf{ID}], T)$
    reward $\neg\mathsf{acc}$ iff $\mathsf{DP}$
    if $\neg\mathsf{acc}$ return   // failure
    $k \leftarrow \mathsf{key}(\gamma, \mathsf{gid})$
    if $\mathsf{ID} \notin \gamma[\mathsf{gid}].G$   // ID removed
      $\mathsf{ep}[\mathsf{gid}, \mathsf{ID}] \leftarrow (-1,-1)$
      reward $k \neq \bot$
    else   // ID in group
      $\mathsf{ep}[\mathsf{gid}, \mathsf{ID}] \leftarrow (t+1, c')$
      if $\mathsf{view}[\mathsf{gid}, t+1, c'] = \bot$
        $\mathsf{view}[\mathsf{gid}, t+1, c'] \leftarrow \gamma$
      else $\mathsf{ViewChecker}(\mathsf{view}[\mathsf{gid}, t+1, c'], \gamma, \mathsf{gid})$
      if $t \neq -1$   // $T$ not output by create
        $\mathsf{VerChanges}(\mathsf{ST}[\mathsf{ID}], \gamma, T[\mathsf{gid}, (t, c), \text{`vec'}, c'])$
    $\mathsf{ST}[\mathsf{ID}] \leftarrow \gamma$

$\mathsf{VerChanges}(\gamma_1, \gamma_2, \vec{P})$
    reward $\mathsf{policy}(\vec{P}, \gamma_1.[\mathsf{gid}].\mathsf{G}) \neq \gamma_2[\mathsf{gid}].\mathsf{G}$
```

Figure 3.1: Correctness game for CGKA with respect to predicates $(\mathsf{OP}, \mathsf{PP}, \mathsf{CP}, \mathsf{DP})$. Lines in gray correspond to liveness predicates.

whether the messages they output are blank ($\perp$) or not. For these algorithms, we require that the state is not changed in the event of failure, a property that we denote by *robustness*. As noted for two-party ratcheting [BSJ+17], badly-formed input should not affect the functionality of the protocol, and in particular should not disrupt protocol execution. Furthermore, robustness simplifies modelling as the effect of failure does not need to be considered.

Liveness. The CGKA algorithms in Definition 3.2 should never fail if the protocol execution is honest and the inputs to the algorithms are well-formed. In particular, this prevents that a trivial protocol (always outputting $\perp$) can satisfy a correctness notion (particularly since a trivial protocol satisfies robustness).

Since the correctness notion can vary depending on the desired behaviour of a CGKA, we defer some checks to external predicates which, in part, parameterize the correctness game. The predicates are included in (**reward** *failure* **iff** P) clauses, marked in gray in Figure 3.2. Namely, the predicates are a necessary and sufficient condition for the CGKA algorithms to succeed. The predicates OP, PP, CP, and DP are described below.

- OP verifies that all the conditions required for successfully creating a group are satisfied. An example for the OP predicate could be $\mathsf{OP} := \mathsf{ID} \in G$, which verifies that the group creator is in the group.
- PP checks the validity of a call to prop. Depending on the envisioned application, it may make sense to enforce that proposals are performed by current group members, or alternatively allow any party to propose group changes.
- CP checks the validity of a call to commit. One possible use of CP is to restrict the users that can perform commits (looking ahead, this may correspond with the group administrators).
- DP checks the validity of a call to proc. One possibility is to verify that the proc algorithm fails whenever a control message T corresponding to a specific group is not delivered to a party which is neither part of the group nor joining it.

Consistency. In a correct CGKA execution, two notions of state consistency should be satisfied: membership consistency and key consistency (closely related to key partnering [PRSS21]). First, we enforce that the group structure can only be modified after a proc algorithm. In particular, we utilize the ViewChecker method to enforce that prop, create and commit update neither the group state nor the group key. We also enforce that all users who transition to the same epoch (after processing a message) have the same view of the group. Furthermore, VerChanges ensures that after a commit is processed, the change in the group structure is coherent with the policy specified in the CGKA for proposals.

New epoch keys are derived upon successful proc calls. Key consistency ensures that that all users who transition to the same epoch derive the same key k, which again is verified using $\mathsf{ViewChecker}$) in the $\mathcal{O}^{\mathsf{Deliver}}$ oracle. The dictionary view stores the state of the first group member ID that moves to a certain epoch $(t+1, c')$; consistency is violated if the state (key and view of the group) of any member in the same epoch is inconsistent with $\mathsf{view}[\mathsf{gid}, t+1, c']$. Moreover, a user who has derived a key $k \neq \perp$ must be a group member, i.e., its epoch should never be $(-1,-1)$.

3.2.4 Capturing Security

The security of a CGKA can be formalized in different ways. One of them is to introduce a game-based adversarial model, such as the KIND security notion for the BARK (Section 2.3). This is the approach followed in works such as [ACDT20, KPPW+21]. Another possibility is to model a CGKA in the Universal Composability (UC) framework [Can01] (see Appendix A) or other related frameworks, which is the approach in [ACJM20, AJM20].

In this thesis, we define CGKA security in Definition 3.3, via the key-indistinguishability security game in Figure 3.2, which is based on [ACDT20]. The game adapts the security game in [ACDT20] to the propose-and-commit paradigm and makes some changes in the notation to unify it with the correctness game in Figure 3.1. We also ignore correctness properties in the game since we have a separate correctness notion (Definition 3.2). Below, we discuss the role of the cleanness predicate, the assumptions on the network (active adversaries) and other desirable properties. Looking ahead, we will refer to Definition 3.3 when discussing the security of TreeKEM (Section 3.3) and the security for the A-CGKA primitive that we introduce in Chapter 4.

Definition 3.3 (Security of CGKA). *A* CGKA *is* (λ, q, ϵ)-$\mathsf{C}_{\mathsf{clean}}$*-secure if, for any polynomial-time adversary* $\mathcal{A}$ *limited to* q *oracle queries, the advantage of* $\mathcal{A}$ *in the* $\mathsf{KIND}_{\mathsf{CGKA}}$ *game (Figure 3.2) given by*

$$\mathsf{Adv}^{\mathsf{kind}_{\mathsf{cgka}}}_{\mathcal{A}}(\lambda) = \left| \Pr[\mathsf{KIND}^{\mathcal{A}}_{\mathsf{CGKA},0,\mathsf{C}_{\mathsf{clean}}}(1^\lambda) = 1] - \Pr[\mathsf{KIND}^{\mathcal{A}}_{\mathsf{CGKA},1,\mathsf{C}_{\mathsf{clean}}}(1^\lambda) = 1] \right|$$

is bounded by ϵ*, where the probability is taken over the choice of the challenger and adversary's random coins.*

Game description. Our security game uses a similar notation to the correctness game in Figure 3.2 (in fact, many lines of the games are common), and expands the notion in [ACDT20] by considering multi-group security[5]. Correctness and security should be read jointly; the adversary must have at least as many capabilities in the

[5]In [ACDT20], a non-deletion scenario is captured, in which the adversary can prevent parties from deleting old secret material. We do not capture this feature since the adversary can repeatedly expose a user on different epochs.

```
$\mathsf{KIND}^{\mathcal{A}}_{\mathsf{CGKA},b,\mathsf{C}_{\mathsf{clean}}}(1^\lambda)$
1 : $\mathsf{K}[\cdot], \mathsf{ST}[\cdot], \mathsf{exp}[\cdot] \leftarrow \bot$
2 : public $\mathsf{T}[\cdot] \leftarrow \bot$
3 : $\mathsf{prop\text{-}ctr}, \mathsf{com\text{-}ctr}, \mathsf{exp\text{-}ctr} \leftarrow 0$
4 : $\mathsf{ep}[\cdot] \leftarrow -1$;   $\mathsf{chall}[\cdot] \leftarrow \mathsf{false}$
5 : $b' \leftarrow\$ \mathcal{A}^{\mathcal{O}}(1^\lambda)$
6 : require $\mathsf{C}_{\mathsf{clean}}$
7 : return $b'$

$\mathcal{O}^{\mathsf{Init}}(\mathsf{ID})$
1 : require $\mathsf{ST}[\mathsf{ID}] = \bot$
2 : $\mathsf{ST}[\mathsf{ID}] \leftarrow\$ \mathsf{init}(1^\lambda, \mathsf{ID})$

$\mathcal{O}^{\mathsf{Create}}(\mathsf{ID}, \mathsf{gid}, G)$
1 : $(\gamma, T) \leftarrow\$ \mathsf{create}(\mathsf{ST}[\mathsf{ID}], \mathsf{gid}, G)$
2 : if $T = \bot$ return   // failure
3 : $\mathsf{com\text{-}ctr} \leftarrow \mathsf{com\text{-}ctr} + 1$
4 : $\mathsf{T}[\mathsf{gid}, -1, \text{‘com’}, \mathsf{com\text{-}ctr}] \leftarrow T$
5 : $\mathsf{ST}[\mathsf{ID}] \leftarrow \gamma$

$\mathcal{O}^{\mathsf{Prop}}(\mathsf{ID}, \mathsf{gid}, \mathsf{ID}', \mathsf{type})$
1 : $(\gamma, P) \leftarrow\$ \mathsf{prop}(\mathsf{ST}[\mathsf{ID}], \mathsf{gid}, \mathsf{ID}', \mathsf{type})$
2 : $\mathsf{prop\text{-}ctr} \leftarrow \mathsf{prop\text{-}ctr} + 1$
3 : $\mathsf{T}[\mathsf{gid}, \mathsf{ep}[\mathsf{gid}, \mathsf{ID}], \text{‘prop’}, \mathsf{prop\text{-}ctr}] \leftarrow P$
4 : $\mathsf{ST}[\mathsf{ID}] \leftarrow \gamma$

$\mathcal{O}^{\mathsf{Commit}}(\mathsf{ID}, \mathsf{gid}, (i_1, \ldots, i_k))$
1 : $\vec{P} \leftarrow (T[\mathsf{gid}, \mathsf{ep}[\mathsf{gid}, \mathsf{ID}], \text{‘prop’}, i])_{i=(i_1,\ldots,i_k)}$
2 : $(\gamma, T) \leftarrow\$ \mathsf{commit}(\mathsf{ST}[\mathsf{ID}], \mathsf{gid}, \vec{P})$
3 : if $T = \bot$   return   // failure
4 : $\mathsf{com\text{-}ctr} \leftarrow \mathsf{com\text{-}ctr} + 1$
5 : $\mathsf{T}[\mathsf{gid}, \mathsf{ep}[\mathsf{gid}, \mathsf{ID}], \text{‘com’}, \mathsf{com\text{-}ctr}] \leftarrow T$
6 : $\mathsf{T}[\mathsf{gid}, \mathsf{ep}[\mathsf{gid}, \mathsf{ID}], \text{‘vec’}, \mathsf{com\text{-}ctr}] \leftarrow \vec{P}$
7 : $\mathsf{ST}[\mathsf{ID}] \leftarrow \gamma$

$\mathcal{O}^{\mathsf{Deliver}}(\mathsf{ID}, \mathsf{gid}, t)$
1 : require $\mathsf{ep}[\mathsf{gid}, \mathsf{ID}] \in \{t, -1\}$
2 : $T \leftarrow \mathsf{T}[\mathsf{gid}, t, \text{‘com’}, c']$   // honest delivery
3 : require $T \neq \bot$
4 : $(\gamma, \mathsf{acc}) \leftarrow \mathsf{proc}(\mathsf{ST}[\mathsf{ID}], T)$
5 : if $\neg\mathsf{acc}$ return   // failure
6 : $k \leftarrow \mathsf{key}(\gamma, \mathsf{gid})$
7 : if $\mathsf{ID} \notin \gamma[\mathsf{gid}].G$   // ID removed
8 :   $\mathsf{ep}[\mathsf{gid}, \mathsf{ID}] \leftarrow -1$
9 : else   // ID in group
10 :   $\mathsf{ep}[\mathsf{gid}, \mathsf{ID}] \leftarrow t + 1$
11 :   if $\mathsf{K}[\mathsf{gid}, t+1] = \bot$
12 :     $\mathsf{K}[\mathsf{gid}, t+1] \leftarrow k$
13 : $\mathsf{ST}[\mathsf{ID}] \leftarrow \gamma$

$\mathcal{O}^{\mathsf{Expose}}(\mathsf{ID})$
1 : $\mathsf{eps}[\cdot] \leftarrow \bot$   // epoch tracker
2 : $\mathsf{exp\text{-}ctr} \leftarrow \mathsf{exp\text{-}ctr} + 1$
3 : for $\mathsf{gid}$ such that $\mathsf{ep}[\mathsf{ID}, \mathsf{gid}] \neq -1$
4 :   $\mathsf{eps}[\mathsf{gid}] \leftarrow \mathsf{ep}[\mathsf{gid}, \mathsf{ID}]$
5 : $\mathsf{exp}[\mathsf{ID}, \mathsf{exp\text{-}ctr}] \leftarrow \mathsf{eps}$
6 : return $\mathsf{ST}[\mathsf{ID}]$   // ID exposed in all groups

$\mathcal{O}^{\mathsf{Reveal}}(\mathsf{gid}, t)$
1 : $\mathsf{chall}[\mathsf{gid}, t] \leftarrow \mathsf{true}$
2 : return $\mathsf{K}[\mathsf{gid}, t]$

$\mathcal{O}^{\mathsf{Challenge}}(\mathsf{gid}, t)$
1 : require $\mathsf{K}[\mathsf{gid}, t] \neq \bot$
2 : $\mathsf{chall}[\mathsf{gid}, t] \leftarrow \mathsf{true}$
3 : if $b = 0$ return $\mathsf{K}[\mathsf{gid}, t]$
4 : if $b = 1$ return $r \leftarrow\$ \{0,1\}^\lambda$
```

Figure 3.2: Security game for CGKA with respect to the $\mathsf{C}_{\mathsf{clean}}$ predicate.

correctness game as in the security game. By separating both sets of properties, we achieve a security notion which is more complete than [ACDT20], since their game *requires* that the group evolves correctly, whereas in our correctness game we *reward* the adversary if this is not the case. Robustness is also not considered in their game.

In our security game, the adversary can interact with a set of oracles. At any point in the game, the adversary can challenge an epoch t on a specific group gid. In a challenge, the adversary is given the group key $\mathsf{K}[\mathsf{gid}, t]$ if the bit that parametrizes the game is $b = 0$, and a random string $r \leftarrow\!\$ \{0,1\}^\lambda$ if $b = 1$. The adversary must try to determine the value of b by outputting a guess b' of b. The game is considered valid when the *cleanness predicate* $\mathsf{C}_{\mathsf{clean}}$ is true. Cleanness ensures that no trivial attacks on the game are possible; we discuss this concept below.

In order to capture the ratcheting of the group keys (PCS and FS), the adversary has two mechanisms to obtain secret group material: it can *expose* a user ID and *reveal* the group secret k associated to a group gid. An exposure leaks the whole state of ID, stored in $\mathsf{ST}[\mathsf{ID}]$, which includes the state of ID in all their groups. In order to keep track of the specific epochs in which ID has been exposed in each group, we use the $\mathsf{exp}[\cdot]$ global variable. On the other hand, a reveal leaks the group key to the adversary on a specified epoch t - in this case, $\mathsf{chall}[\mathsf{gid}, t]$ is set to true to prevent the adversary from challenging such an epoch (this restriction needs to be captured in the security game).

Cleanness predicate. As with the BARK in Section 2.3, the cleanness predicate $\mathsf{C}_{\mathsf{clean}}$ plays a double role. First, it excludes the trivial attacks on the protocol, i.e., those that break security unavoidably such as revealing the group secret and issuing a challenge in the same epoch. Second, it captures the exact security of the protocol, in our case, referring to PCS and FS. Therefore, our cleanness predicate can be expressed as $\mathsf{C}_{\mathsf{clean}} = \mathsf{C}_{\mathsf{opt}} \wedge \mathsf{C}_{\mathsf{add}}$, where $\mathsf{C}_{\mathsf{opt}}$ is an *optimal*, generic cleanness predicate that excludes only trivial attacks, and $\mathsf{C}_{\mathsf{add}}$ is an additional cleanness predicate that depends on the scheme and excludes other types of attacks. We define $\mathsf{C}_{\mathsf{opt}}$ in a similar way to the safe predicate in [ACDT20]. Namely, we exclude the following cases (for a specific group gid): (i) the group secret in the challenge epoch t^* has already been challenged or revealed, and (ii) a group member ID whose state was exposed in epoch $t < t^*$ has not updated their keys (i.e., performed a commit, or been involved in an add, remove, or update proposal) or been removed before the challenge epoch t^*. Formally, for an adversary that does $q_1, \ldots, q_q$ oracle queries in the game in Figure 3.2, the optimal cleanness predicate is given by:

$$\begin{aligned}\mathsf{C}_{\mathsf{opt}} = \bigwedge_{\mathsf{gid}} &\big(\forall i : q_i = \mathcal{O}^{\mathsf{Challenge}}(\mathsf{gid}, t^*), \mathsf{chall}[\mathsf{gid}, t^*] = \mathsf{false}\big) \wedge \\ &\big(\forall (i, \mathsf{ID}, \mathsf{exp\text{-}ctr}) : q_i = \mathcal{O}^{\mathsf{Challenge}}(\mathsf{gid}, t^*), \exists t : (0 \le \mathsf{tExp}(\mathsf{ID}, \mathsf{exp\text{-}ctr}) < t \le t^*) \wedge \\ &\mathsf{hasUpd}\left(\mathsf{ID}, T[\mathsf{gid}, t, \mathsf{com}, \cdot], T[\mathsf{gid}, t, \mathsf{vec}, \cdot]\right)\big).\end{aligned}$$

where we have used the following auxiliary functions. The function $\mathsf{tExp}(\mathsf{ID}, \mathsf{exp\text{-}ctr}) = \mathsf{exp}[\mathsf{ID}, \mathsf{exp\text{-}ctr}][\mathsf{gid}]$ if $\exists k : q_k = \mathcal{O}^{\mathsf{Expose}}(\mathsf{ID})$, and $\mathsf{tExp} = -1$ otherwise. The function $\mathsf{hasUpd}(\mathsf{ID}, T, \vec{P})$ outputs true if either: (i) ID has commited the message T, or (ii) ID has been included in an add, update, or removed proposal in $\vec{P}$.

We note that the condition on the reveal oracle is very strong, since an adversary that reveals the group secret on epoch t can still challenge any epoch $t^* \neq t$.

Limitations. Our security definition restricts the capabilities of the adversary since it captures passive adversaries, which corresponds to network eavesdroppers. For CGKAs, there exists work in the literature that provides security against active adversaries who can schedule messages arbitrarily [KPPW+21] or inject messages [ACJM20] (in the UC model). Among the broader family of group key agreement protocols, where long-lived sessions and PCS are generally not considered, modelling active adversaries is standard practice [PRSS21].

Furthermore, our game does not consider insider adversaries, i.e., malicious group members that can deviate from protocol execution [KS05], [AJM20]. We also do not model authentication and randomness manipulation.

3.3 TreeKEM and Variants

The MLS protocol[6] implements a CGKA construction named TreeKEM [BBR18]. TreeKEM is based on binary ratchet trees, which were already in use in early versions of MLS who implemented the so-called Asynchronous Ratchet Trees [CGCG+18]. Tree-based protocols are an attractive choice for messaging since they achieve average-case $O(\log n)$ complexity in most group operations. Several variants of TreeKEM and ARTs have been proposed in the literature, such as Tainted TreeKEM [KPPW+21], Re-randomized TreeKEM [ACDT20], Casual TreeKEM [Wei19], and Tripartite ARTs [Gaj20].

In TreeKEM, group members share a ratchet tree τ, where each user is associated with a single leaf. Each tree node v has an assigned PKE key pair $(v.\mathsf{sk}, v.\mathsf{pk})$, except for the root, which holds the epoch secret I known by all group members, and except for some intermediate nodes that can be blank. At all times, the tree must satisfy two invariants.

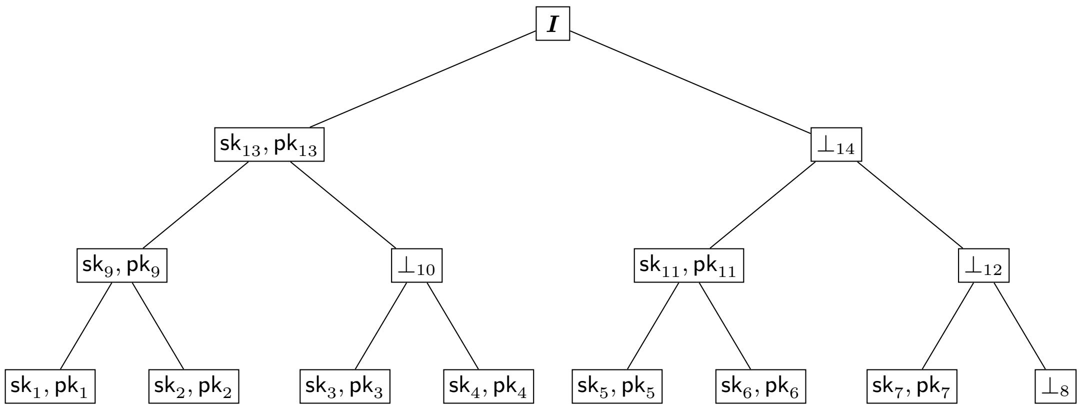
1. Each user knows all secret keys on its direct path, which is the path from its leaf node to the root. Equivalently, the secret keys of a node v are known only to users in the subtree rooted at v, as seen in Figure 3.3.
2. Each user knows all public key material in the tree τ.

[6]At the time of writing, the MLS draft is at version 11 [BBM+]

Besides, all members must have the same view of the tree after processing the same updates. The main operations in TreeKEM are member additions, member removals, and key updates. The current version of TreeKEM (MLSv11) [BBM+] uses the propose-and-commit paradigm, following Definition 3.1.

3.3.1 Ratchet Trees

We present the main parts of the description of TreeKEM in [ACDT20]. The ratchet trees in TreeKEM are left-balanced binary trees (LBBTs) of n nodes, meaning that all leaves are located at depths $d+1$ and d, and that all leaves situated at the left of any label at depth $d+1$ are also at depth $d+1$. Every node is labeled; the root node is labeled with the epoch secret I, the internal nodes v with the pair $(v.\mathsf{sk}, v.\mathsf{pk})$, and the leaves v with the triple $(v.\mathsf{sk}, v.\mathsf{pk}, v.\mathsf{ID})$. A node may also be blank, meaning that its labels are set to $\perp$.



I
$\mathsf{sk}_{13}, \mathsf{pk}_{13}$
$\perp_{14}$
$\mathsf{sk}_9, \mathsf{pk}_9$
$\perp_{10}$
$\mathsf{sk}_{11}, \mathsf{pk}_{11}$
$\perp_{12}$
$\mathsf{sk}_1, \mathsf{pk}_1$
$\mathsf{sk}_2, \mathsf{pk}_2$
$\mathsf{sk}_3, \mathsf{pk}_3$
$\mathsf{sk}_4, \mathsf{pk}_4$
$\mathsf{sk}_5, \mathsf{pk}_5$
$\mathsf{sk}_6, \mathsf{pk}_6$
$\mathsf{sk}_7, \mathsf{pk}_7$
$\perp_8$

Figure 3.3: Diagram of a TreeKEM-based ratchet tree for a group with 7 members. Blank nodes are represented by $\perp_i$. The leaves $v_1, \ldots, v_8$ correspond to group members $\mathsf{ID}_1, \ldots, \mathsf{ID}_7$ respectively (notice that v_8 is blank). The direct path of node v_1 is formed by nodes v_9, v_{13}, and the root; hence ID_1 knows the tree secrets $\mathsf{sk}_1, \mathsf{sk}_9, \mathsf{sk}_{13}, I$.

The *resolution* of a node v is the minimal subset of non-blank nodes that covers the whole subtree rooted at v. For example, in Figure 3.3, the resolution of the node v_{14} is the set $\{v_7, v_{11}\}$. The resolution only differs from v in case v is a blank node; in this case, one must dive recursively into the tree until a covering of non-blank nodes is found.

Tree Operations. Given a list of users $G = (\mathsf{ID}_1, \ldots, \mathsf{ID}_n)$ and public keys $(\mathsf{pk}_1, \ldots, \mathsf{pk}_n)$ where the creator ID_1 has a secret key sk_1, the corresponding LBBT of n nodes is *initialized* blank and *created* with a single node labeled as $(\mathsf{sk}_1, \mathsf{pk}_1, \mathsf{ID}_1)$. The remaining members $(\mathsf{ID}_2, \ldots, \mathsf{ID}_n)$ are then added by committing add proposals.

For *adding* a user ID_i, the algorithm first checks whether there are any blank leaves in the tree. In that case, one of them is picked and labeled $(\perp, \mathsf{pk}_i, \mathsf{ID}_i)$. Otherwise, a new leaf is added. For *removing* a user ID_i, its corresponding leaf is blanked. Then, all the ancestors of the leaf (i.e., its direct path to the root) are blanked too, including the root itself. Notice that this erases all secrets known by ID_i according to invariant 1 (as seen in Figure 3.4). The tree is then truncated such that the rightmost assigned leaf is the last node of the tree.

3.3.2 TreeKEM as CGKA

The core of TreeKEM builds upon two cryptographic primitives that we have introduced implicitly: an IND-CPA secure PKE $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ and a pseudorandom generator PRG[7]. We introduce the protocol operations of TreeKEM, following [BBM+, ACDT20, AJM20], and the CGKA abstraction in Definition 3.1. We note that MLS builds on TreeKEM and supports more operations, such as external proposals, re-initializations or acknowledgements. We also ignore the role of the gid below since multi-groups are not modelled in [ACDT20, AJM20].

- $\gamma \leftarrow \mathsf{init}(1^\lambda, \mathsf{ID})$ sets the security parameter $\gamma[\cdot].\lambda \leftarrow \lambda$, the variable $\gamma[\cdot].\mathsf{ME} \leftarrow \mathsf{ID}$ that specifies the ID of the party running a local TreeKEM instance, and initializes an empty ratchet tree $\gamma[\cdot].\tau \leftarrow \perp$.

- $(\gamma', T) \leftarrow \mathsf{create}(\gamma, \mathsf{gid}, G)$ only accepts the case in which G contains a single ID (which must be $\gamma[\cdot].\mathsf{ME}$). The control message T is blank, and the group creator automatically processes the group creation. The ratchet tree $\gamma[\mathsf{gid}].\tau$ contains a single node.

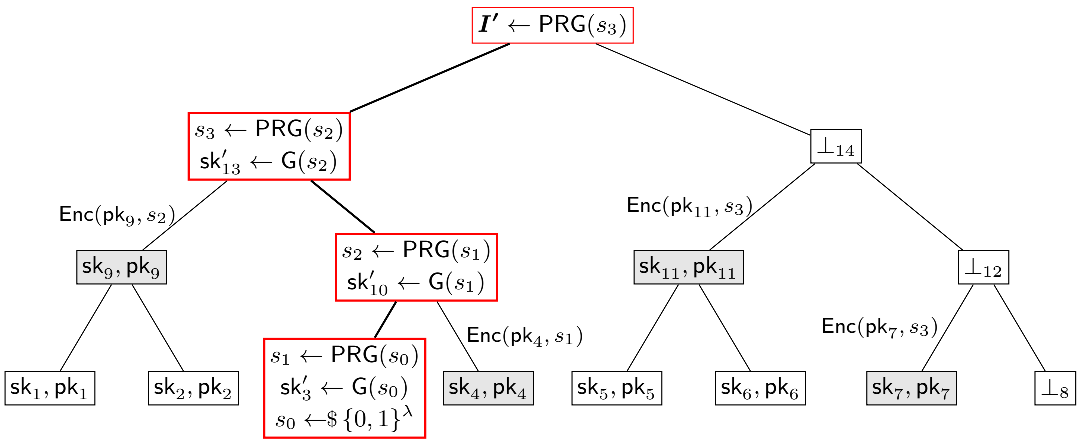
- $P \leftarrow \mathsf{prop}(\gamma, \mathsf{gid}, \mathsf{ID}, \mathsf{type})$ generates an add, remove, or update proposal P (specified by type) which is broadcast to the whole group, and not applied until they are committed by another member (the state is not updated, i.e., $\gamma' = \gamma$). An add proposal is created by including the new member ID's key package $(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}})$, fetched from the Delivery Service. The position of the new member in the ratchet tree is later decided by the committer; this will be a blank leaf if there are any in the tree, or a new leaf. A rem proposal specifies the position in the tree of the member ID to be removed. The committer will blank all nodes on the direct path of ID, meaning that ID should no longer know any group secrets after that. An upd proposal includes a new key package $(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}})$ for the sender ID. When it is committed, the key pair on the leaf v assigned to ID is replaced by $(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}})$ and all remaining nodes in ID's direct path are blanked (as for a removed user). Hence all the secrets previously known by ID are erased from the tree, allowing ID to recover from a state compromise.

- $(\gamma', T) \leftarrow\$ \mathsf{commit}(\gamma, \mathsf{gid}, \vec{P})$ advances the group to the next epoch. The commit consists in applying all valid proposals $(P \in \vec{P})$ that the committer has received

[7]In practice, pseudorandom generators are implemented using some cryptographic primitive. One example is the HKDF used in [AJM20], which is a KDF based on a hash-based MAC (HMAC).

during the current epoch, according to policy. After applying the proposals, the committer must perform a *full update* operation, which consists on refreshing all keys in the direct path of the committer up to the root, filling in blank nodes if necessary. This is done in the following way. First, the committer randomly samples a fresh *path secret* s_0 and derives a sequence $s_i = \mathsf{PRG}(s_{i-1})$ of secrets for all nodes in his direct path. Second, from each s_i, the committer derives a node secret $s'_i = \mathsf{PRG}(s_i)$. Third, the node secrets are used for deterministic key generation; $(\mathsf{pk}_i, \mathsf{sk}_i) \leftarrow \mathsf{Gen}(s_i)$. Fourth, the node secret s_{root} for the root node is the new epoch secret $I = s_{\text{root}}$.

An example of a full update operation is shown in Figure 3.4. A commit can also be proposal free. In this case, the only operation that is performed is the full update.



$\boldsymbol{I'} \leftarrow \mathsf{PRG}(s_3)$
$s_3 \leftarrow \mathsf{PRG}(s_2)$
$\mathsf{sk}'_{13} \leftarrow \mathsf{G}(s_2)$
$\perp_{14}$
$\mathsf{Enc}(\mathsf{pk}_9, s_2)$
$\mathsf{Enc}(\mathsf{pk}_{11}, s_3)$
$\mathsf{sk}_9, \mathsf{pk}_9$
$s_2 \leftarrow \mathsf{PRG}(s_1)$
$\mathsf{sk}'_{10} \leftarrow \mathsf{G}(s_1)$
$\mathsf{sk}_{11}, \mathsf{pk}_{11}$
$\perp_{12}$
$\mathsf{Enc}(\mathsf{pk}_4, s_1)$
$\mathsf{Enc}(\mathsf{pk}_7, s_3)$
$\mathsf{sk}_1, \mathsf{pk}_1$
$\mathsf{sk}_2, \mathsf{pk}_2$
$s_1 \leftarrow \mathsf{PRG}(s_0)$
$\mathsf{sk}'_3 \leftarrow \mathsf{G}(s_0)$
$s_0 \leftarrow\!\$ \{0,1\}^\lambda$
$\mathsf{sk}_4, \mathsf{pk}_4$
$\mathsf{sk}_5, \mathsf{pk}_5$
$\mathsf{sk}_6, \mathsf{pk}_6$
$\mathsf{sk}_7, \mathsf{pk}_7$
$\perp_8$

Figure 3.4: Diagram of a TreeKEM full update operation triggered by ID_3, based on the tree in Figure 3.3. The full update is started with the generation of a random seed s_0, the derivation a node secret key sk'_3, and the update of the nodes on its direct path (v_{10}, v_{13}, and the root) accordingly. The function $\mathsf{G} = \mathsf{Gen} \circ \mathsf{PRG}$ derives first the node secret, and then the new key pair of the node (the public key is omitted for conciseness).
The path secrets s_1, s_2, s_3 are encrypted under the public keys of the resolution of the sibling nodes v_4, v_9, v_{14}, which is the set $\{v_4, v_7, v_9, v_{11}\}$ marked in grey (the resolution of the blank node v_{14} consists of two nodes). Notice that every group member knows a secret key that allows them to recover the path secrets.

As a result of a commit, a commit message T is broadcast to all group members. The message includes the encryption of each of the path secrets s_i under the public keys of the resolution of the sibling nodes changed in the commit, as seen in Figure 3.4. This ensures that every member can recover one of the s_i. Besides, it includes the proposal vector $\vec{P}$ and metadata required to verify the correctness of the commit and to update their local copy of the ratchet tree. The message T also contains a copy of the ratchet tree for incoming members, which is encrypted under the key pairs specified in the add proposals in $\vec{P}$.

- $(\gamma', \mathsf{acc}) \leftarrow \mathsf{proc}(\gamma, T)$ makes effective all the changes introduced in the group by a commit, upon reception of a control message T. Let ID be the member that receives T. If ID is in the group, it will be able to decrypt one of the path secrets s_i. Then, ID must derive the full chain of secrets $s_{i+1} = \mathsf{PRG}(s_i)$ starting from s_i and the key pairs $(\mathsf{sk}_i, \mathsf{pk}_i) \leftarrow \mathsf{Gen}(s_i)$. The result is an updated version of the ratchet tree which is the same among all group members. If ID is not in the group and T includes welcome information for ID, then ID stores a local copy of the ratchet tree according to T.
- $k \leftarrow \mathsf{key}(\gamma, \mathsf{gid})$ simply returns the epoch secret $k = I$ corresponding to the calling user's state $\gamma[\mathsf{gid}]$.
- $G' \leftarrow \mathsf{policy}(\vec{P}, G)$ establishes that in the case of proposals concerning the same node, remove proposals have the highest priority, followed by the most recent update proposal. We note that this function is not defined explicitly in the literature.

3.3.3 Standard TreeKEM

The standard version of TreeKEM refers to the core CGKA presented above. Alwen et. al. performed the first formal analysis of the security of TreeKEM [ACDT20], which shows that PCS is achieved as desired, but FS is only achieved in a weak form. Their security model corresponds to the game in Figure 3.2, i.e., considers passive and non-adaptive adversaries[8] and does not consider randomness exposure or manipulation.

More specifically, TreeKEM is proven secure with respect to $\mathsf{C}_{\mathsf{clean}} = \mathsf{C}_{\mathsf{opt}} \wedge \mathsf{C}_{\mathsf{weakFS}}$, which is the conjunction of the optimal predicate $\mathsf{C}_{\mathsf{opt}}$ defined in Section 3.2.4, and the following weak forward secrecy predicate[9] (adapted to the multi-group setting). The $\mathsf{C}_{\mathsf{weakFS}}$ predicate ensures that, if ID is a group member whose last update was done in epoch t, and ID is compromised in a later epoch tExp, then all challenges in epochs t^* such that $t \leq t^* < \mathsf{tExp}$ are excluded. Note that this represents a weak form of FS, since we restrict challenges on epochs *before* a state exposure.

$$\begin{aligned}\mathsf{C}_{\mathsf{weakFS}} = \bigwedge_{\mathsf{gid}} \big(\forall (i, \mathsf{ID}, \mathsf{exp\text{-}ctr}) : q_i = \mathcal{O}^{\mathsf{Challenge}}(\mathsf{gid}, t^*), \\ \exists t : (0 \leq t^* \leq t < \mathsf{tExp}(\mathsf{ID}, \mathsf{exp\text{-}ctr})) \wedge \\ \mathsf{hasUpd}\,(\mathsf{ID}, T[\mathsf{gid}, t, \mathsf{com}, \cdot], T[\mathsf{gid}, t, \mathsf{vec}, \cdot])\,\big).\end{aligned}$$

Note that PCS is already captured in $\mathsf{C}_{\mathsf{clean}}$; if ID is a group member which is compromised during epoch tExp and updates in epoch $t > \mathsf{tExp}$, all challenges in epochs t^* such that $\mathsf{tExp} \leq t^* < t$ are already excluded.

[8]A proof with respect to adaptive adversaries is also presented, but there is a subexponential loss of tightness (in the group size).

[9]In [ACDT20], TreeKEM is also proven secure with respect to a slightly stronger predicate named tkm, which is more involved.

The loss of forward security occurs because the path secrets are encrypted under key pairs that are known by ID, in particular under their leaf key pair $(\mathsf{sk}_{\mathsf{ID}}, \mathsf{pk}_{\mathsf{ID}})$, which is not updated since epoch t. If the adversary learns ID's private key in epoch t', it will be able to decrypt all path epochs encrypted under $\mathsf{pk}_{\mathsf{ID}}$ between epochs t and t', potentially leading to the recovery of a group secret from an earlier epoch.

In [ACDT20], this issue is solved by replacing the PKE by Updatable-PKE (more specifically, HkuPKE). Then, the key pair $(\mathsf{sk}_i, \mathsf{pk}_i)$ associated to a node i is refreshed every time a secret s_i is encrypted at node i, including encryptions at the leaves. We note that the fix is still insufficient against adversaries who can manipulate randomness.

Fixing attempts. A trivial approach to fix the forward security of TreeKEM is the introduction of a symmetric ratchet key schedule such that the secrets I are not the final keys, but rather intermediate values. Namely, let S_t be a secret value corresponding to epoch t, that we compute as

$$S_t = \mathsf{KDF}(S_{t-1}, I_t),$$

where I_t is the TreeKEM secret on epoch t. Then, one can build a sequence of secrets of the form:

$$\begin{tikzcd}
 & I_t \arrow[d] & I_{t+1} \arrow[d] & I_{t+2} \arrow[d] & \\
S_{t-1} \arrow[r] & S_t \arrow[r] & S_{t+1} \arrow[r] & S_{t+2} \arrow[r] & \cdots
\end{tikzcd}$$

where the S_t are immediately deleted after every epoch. Suppose that an adversary compromises a group member on epoch $t+2$, learning the secrets I_{t+2}, S_{t+2}. Due to the weak forward security of TreeKEM, the adversary may be able to learn previous values of I, such as I_{t+1}. However, since the secret S_{t+1} is calculated as $\mathsf{KDF}(S_t, I_{t+1})$, the knowledge of I_{t+1} is no longer sufficient for the adversary, who also requires S_t to derive S_{t+1}. The secrets S remain forward secret unless the adversary learns the starting point of the ratchet, and they remain post-compromise secure since they are derived from the TreeKEM secrets I.

Nevertheless, there is still an attack available to the adversary. Suppose that two (not necessarily distinct) users A and B are compromised in epochs t and $t+2$, respectively. Additionally, suppose that in epoch $t+1$, A updates. The adversary then learns $S_t, I_t, S_{t+2}, S_{t+2}$, but not S_{i+1}. Due to the lack of forward security of TreeKEM, the compromise of B may allow the adversary to recover I_{t+1}, which easily yields S_{t+1} using the above data.

Hence, even if the security is strictly better if we derive the group secrets as above (which is already done in the MLS key schedule), forward security is not guaranteed after a first compromise. This precise security notion is sometimes denoted by PCFS (post-compromise forward security) in the literature [AJM20].

3.3.4 Tainted TreeKEM

The blanking of intermediate nodes in TreeKEM every time a user updates or is removed is required for security, but it increases the communication complexity of the protocol. Indeed, the path secrets after a commit will be encrypted under the resolution of the nodes in the co-path; if some of these are blank, their resolution will consist of more than one node, increasing the number of encryptions needed. While it is true that nodes will heal after a full update, the impact on performance can be significant (namely, the number of encryptions per update degrades from $\mathcal{O}(\log n)$ to $\mathcal{O}(n)$).

In the Tainted TreeKEM protocol, nodes are no longer blanked, but *tainted* [KPPW+21]. A node can be marked as tainted by one or more group members, meaning that they know the secret key material of the node despite not being on their direct path to the root. As long as members keep track of the tainting of the tree nodes, Tainted TreeKEM can be proven secure. The main advantage is that the absence of blank nodes ensures that a balanced binary tree structure is maintained. On the other hand, tainted nodes need to be overwritten and some operations become more expensive. The efficiency of Tainted TreeKEM is difficult to compare to TreeKEM as they perform better in different scenarios; for example, Tainted TreeKEM is noticeably faster if only a small subset of the members (such as group administrators) perform most of the commit operations.

The main motivation for Tainted TreeKEM is therefore efficiency. Nevertheless, the work in [KPPW+21] presents several remarkable improvements in the security model with respect to [ACDT20] as presented below.

- **Partial active security** is achieved by modelling an adversary which can schedule the messages of the Delivery Service arbitrarily. In fact, the semantics of Tainted TreeKEM differ from our definition of CGKA, since control messages are processed temporarily and confirmed using a confirm-and-deliver algorithm. Nevertheless, the adversary cannot use secrets of corrupted parties to forge messages and deviate from the protocol.

- **Adaptive security** is proven both in the Standard Model and in the Random Oracle Model, using different proof techniques. For a group with up to n total members over time and Q oracle queries of the adversary, they achieve a $\mathcal{O}\big(n^2 Q^{\log n}\big)$ tightness in the standard model and a $\mathcal{O}\big(n^2 Q^2\big)$ tightness in the random oracle model.

- **Randomness exposure** is considered, in the sense that the $\mathcal{O}^{\mathsf{Expose}}(\mathsf{ID})$ oracle gives the adversary access to the randomness used by ID in group operations. The adversary cannot, however, manipulate such randomness.

The construction of the cleanness predicate is more involved since users may have different views of who is a group member, due to the adversarial control of the Delivery Service. For this purpose, a *critical window* is defined for each member ID, which models the epoch interval in which ID can be corrupted without leaking the group key, reflecting FS and PCS properties (such as in the previous section). A game is considered clean if, for any challenge on a group key k resulting from a commit by ID, no member ID^* is (or has been) corrupted on their critical window according to the view of the protocol from ID.

There are also some important drawbacks of the security analysis in [KPPW+21], besides the lack of randomness manipulation and truly active adversaries. For instance, a single group is allowed, and the adversary is not given access to an $\mathcal{O}^{\mathsf{Reveal}}$ oracle for learning the group key; it can only corrupt individual parties. These shortcomings generate composability issues, according to [PRSS21].

3.3.5 Insider-Secure TreeKEM

The current draft of MLS includes several mechanisms, which are integrated on top of TreeKEM, that improve the robustness of the protocol against users that deviate from normal protocol execution. Such mechanisms compose the so-called Insider-Secure TreeKEM protocol and include confirmation tags, packet MACing and signing, and *tree signing*. These techniques improve the so-called *insider security* of MLS, as studied in [AJM20]. The concept of insider security [KS05] pursues two main objectives: capturing security for corrupt group members (i.e. malicious insiders), and providing security for new group members.

Malicious insiders. The adversarial model in [AJM20] captures an active adversary with the additional capability of registering corrupted key packages in the PKI (Delivery Service and Authentication Service), as a corrupt member might do. The cleanness predicate that results from such behaviour is very complex, since the key partnering is more involved. For example, if a group member is not compromised but uses a corrupt key package in a commit, other keys may be leaked. Besides, all packets are signed by the senders; if the randomness used in a signature is exposed, the private signature keys may leak if certain signature schemes are used (such as ECDSA).

Security for new members: tree signing. During any commit in MLS, the committing user signs every key refreshed in the full update under their long-term identity key. In this way, a new member Bob can verify that the keys in the

TreeKEM nodes were honestly generated by a group member (as long as the long-term identity key remains secure). Hence, a malicious insider Charlie should not be able to craft an artificial group (with a different TreeKEM configuration) in which, if Bob removes Charlie, there are remaining secrets known to Charlie. As proven in [AJM20], the current tree signing is insufficient for this purpose; stronger alternatives are not included in MLS since they would provoke a loss of deniability.

Weaknesses. Even though the security model in [AJM20] is proposed in the UC framework [Can01] (see Appendix A), multi-group security does not follow even if the PKI is modelled as a global functionality. Hence, their use of the UC framework does not suffice to ensure secure composability. Besides, the fact that insider adversaries are captured in the security game does not imply that their impact is mitigated. Indeed, the loss of security caused by an insider is basically complete until: 1) all group members behave honestly again, 2) all members recover from state compromise, including randomness, and 3) corrupted key packages are no longer stored or used. Hence, we believe that there is large room for improvement on the insider security of MLS. One step in this direction is to include protections against active attacks [ACJM20]; another step is to provide the group with a robust administration mechanism as we suggest in Chapter 4.

Chapter 4

Group Administrators

There is a strong trend in practice to distinguish between two types of users in a group: administrators and standard users. A group administrator (or simply admin) has all the capabilities of a standard user plus a set of exclusive administrative rights. These often include the ability to add and remove participants, modify group descriptions, approve add requests, delegate their functionality to other admins, etc. As we motivate below, utilizing administrators in which the administrative properties are enforced through cryptography can provide significant improvements in the security of group messaging.

In this chapter, we motivate and introduce a new primitive, the administrative CGKA (A-CGKA), which extends the CGKA primitive to support secure administration. Further, we present two different A-CGKA constructions: *individual admin signatures* (IAS) and *dynamic group signature* (DGS). In IAS, we construct an A-CGKA on top of a CGKA by adding additional administration mechanisms, based on authentication via signatures. In DGS, we employ two independent but synchronized CGKAs. Both approaches were selected due to their simplicity, efficiency and flexibility to adapt to the underlying CGKAs.

4.1 Motivation

In commercial messaging services, group administrators are currently implemented at the application level via policies enforced by either the central server or the user. Examples are the popular messaging apps WhatsApp [Wha16], Telegram, and Signal Messenger (as of 2021).

- In WhatsApp, only group administrators can add and remove users, create and share a group invite link, designate other admins, and remove admin privileges from other admins (except for the group creator, who always remains as an admin). There must be at least one admin in the group; when the last admin leaves, a user is selected randomly as the new admin.

- In Telegram, the group creator is labeled as the *owner*. The owner is a 'super-admin' that can designate other admins with diverse sets of capabilities. Besides adding and removing users, admins can impose partial bans on any user's capabilities, such as sending or receiving messages, and even restricting the content that users can send [Tel]. Such "fine-grained administration" is only possible if end-to-end encryption is disabled; by default, Telegram uses a central server that decrypts all messages.
- In Signal Messenger, admins can specify whether all members or only some of them can add and remove users from a group and create a group invite link.

Despite being widely extended in practice, there is little mention of admins in the literature. Protocols such as MLS or Signal, and GKE abstractions (including CGKA and DCGKA) make no formal distinction between admin and non-admin users, which results in giving admin capabilities to all users. In [RMS18], admins are mentioned as part of the group protocols, but are not characterized formally. In Tainted TreeKEM [KPPW$^+$21], there are efficiency advantages if there is a reduced number of users with administrative capabilities.

Capabilities. Let $G = \{\mathsf{ID}_1, \ldots, \mathsf{ID}_n\}$ be a group of users in a messaging or key exchange protocol and $G^* \subseteq G$ be a non-empty subset of group administrators. The administrators $\mathsf{ID} \in G^*$ that we consider have the following additional capabilities with respect to standard users:

- add and remove members from the group,
- approve and decline join and removal requests,
- designate other group administrators,
- give up its own admin status, and
- remove the admin status from another user.

Other properties found in messaging apps such as changing the group description, establish a pinned post, or sharing invite links are not considered initially, but are discussed at the end of this chapter.

Security goals. There are four main security goals that cryptographic administrators aim to achieve.

1. Reduce the trust put on central servers that coordinate message delivery. The server no longer controls of the list of admins, and potentially should not even know their identity.

2. Mitigate the impact of insider attacks [KS05, AJM20] on protocol execution.

3. Reduce concurrency issues (see Section 3.2.3) if there is no global ordering of messages [WKHB20], since only a reduced set of members are able to commit group changes.

4. Increase the robustness of implementations of messaging protocols, preventing pitfalls such as the *burgle into a group attack* in [RMS18], which is a vulnerability that allows an adversary to enter a group given a partial control of the central server. This particular issue affected group chats in Signal and WhatsApp.

In decentralized groups, achieving such properties is even more important. One reason is that the impact of an insider adversary can be larger due to the lack of a central server that broadcasts messages. For example, a malicious insider can invite a member to the group and only notify a subset of the users, or can send fake commits or proposals. Besides, concurrency issues are intrinsic to these protocols, and so admins can help mitigate them.

4.2 Administrated Continuous Group Key Agreement

To capture the functionality of an administrated messaging group, we introduce the concept of an *administrated continuous group key agreement* (A-CGKA). Just as a CGKA is a suitable primitive to support dynamic ratcheted group key exchange, the A-CGKA is an extension of CGKA that provides administrative functionality. In this section, we describe this new primitive, present a correctness game and discuss possible security notions.

4.2.1 Definition

In an A-CGKA, we denote the subset of group administrators by $G^* \subseteq G$. An admin $\mathsf{ID} \in G^*$ can commit (and therefore make effective) changes to the group structure, such as adding and removing users. As with G in both CGKA and A-CGKA, the subgroup G^* of administrators is dynamic - only admins should be able to commit changes in G^*. A relevant property of A-CGKAs, as defined below, is that administrators can perform updates on some administrative key material. This feature provides native support to constructions which provide PCS with respect to the administration keying material.

Definition 4.1. *An administrated continuous group key agreement (A-CGKA) scheme is a tuple of algorithms* $\mathsf{A\text{-}CGKA} = (\mathsf{init}, \mathsf{create}, \mathsf{prop}, \mathsf{commit}, \mathsf{proc}, \mathsf{key}, \mathsf{policy})$ *such that:*

- *Algorithms* $\mathsf{init}, \mathsf{proc}, \mathsf{key}$ *are defined as for a* CGKA *(Definition 3.1).*

- *In* prop*,* types *is redefined as* $\mathsf{types} = \{\mathsf{add}, \mathsf{rem}, \mathsf{upd}, \mathsf{add\text{-}adm}, \mathsf{rem\text{-}adm}, \mathsf{upd\text{-}adm}\}$*.*
- $(\gamma', T) \leftarrow\$\, \mathsf{create}(\gamma, \mathsf{gid}, G, G^*)$ *additionally takes a subgroup of admins* $G^* \subseteq G$*.*
- $(G', G^{*\prime}) \leftarrow \mathsf{policy}(\vec{P}, G, G^*)$ *additionally takes a set of* ID*s* G^* *(admins) and additionally outputs a set* $G^{*\prime}$*.*
- $(\gamma', T) \leftarrow\$\, \mathsf{commit}(\gamma, \mathsf{gid}, \vec{P}, \mathsf{com\text{-}type})$ *additionally takes a commit type* $\mathsf{com\text{-}type} \in \mathsf{com\text{-}types} = \{\mathsf{std}, \mathsf{adm}, \mathsf{both}\}$*.*

Given ID*'s state* γ *and* gid*,* $\gamma[\mathsf{gid}].G$ *is defined as in Definition 3.1, and* $\gamma[\mathsf{gid}].G^*$ *stores the set of admins in* gid *from* ID*'s perspective.*

Semantics. The execution of an A-CGKA is analogous to that of a CGKA. Besides the introduction of the admin subgroup, we introduce three more proposal types add-adm, rem-adm, and upd-adm, which concern administrative changes. Namely, an admin may add another admin to the group of administrators, revoke the admin capabilities from a party, or update their administrative key material. The commit type com-type specifies the extent of a commit operation; that is, whether it affects the general group (std), the administration of the group (adm), or both at the same time (both). For the latter, a simple example is when an admin is both adding a member (group modification) and refreshing their admin keys (admin modification). We also extend the functionality of policy to consider the effects of commit messages on the admins of a group (in addition to regular group members).

Our create algorithm enforces the condition $G^* \subseteq G$, and our correctness notion will ensure that this condition holds throughout protocol execution.

4.2.2 Correctness

Our proposal for correctness of an A-CGKA is defined in an analogous way as for the CGKA in Definition 3.2. In Figure 4.1, we introduce the correctness game, highlighting the changes with respect to the $\mathsf{CORR}^{\mathcal{A}}_{\mathsf{CGKA}}$ game in Figure 3.1.

Definition 4.2 (Correctness of A-CGKA)**.** *Let* $(\mathsf{OP}, \mathsf{PP}, \mathsf{CP}, \mathsf{DP})$ *denote a tuple of predicates. An* A-CGKA *is* $(\mathsf{OP}, \mathsf{PP}, \mathsf{CP}, \mathsf{DP})$*-correct if, for all* λ *and all computationally unbounded adversaries* $\mathcal{A}$*, it holds that:*

$$\Pr[\mathsf{CORR}^{\mathcal{A}}_{\mathsf{CGKA}}(1^\lambda) = 1] = 0$$

where the probability is taken over the choice of the challenger and adversary's random coins.

Figures 3.2 and 4.2 are identical with the exception of a few lines that concern the correctness of the group administration. Therefore, we refer the reader to Section 3.2.3 for a detailed description of the game. The most relevant addition with respect

```
$\mathsf{CORR}^{\mathcal{A}}_{\mathsf{A\text{-}CGKA}}(1^\lambda)$
1 : public $\mathsf{view}[\cdot], \mathsf{ST}[\cdot], \mathsf{T}[\cdot] \leftarrow \perp$
2 : public $\mathsf{used}[\cdot] \leftarrow \mathsf{false}$   // group id's
3 : public $\mathsf{prop\text{-}ctr}, \mathsf{com\text{-}ctr} \leftarrow 0$
4 : public $\mathsf{ep}[\cdot] \leftarrow (-1,-1)$   // not in group
5 : $\mathcal{A}^{\mathcal{O}}(1^\lambda)$
6 : return 0
```

```
$\mathcal{O}^{\mathsf{Init}}(1^\lambda, \mathsf{ID})$
1 : require $\mathsf{ST}[\mathsf{ID}] = \perp$
2 : $\mathsf{ST}[\mathsf{ID}] \leftarrow\$ \mathsf{init}(1^\lambda, \mathsf{ID})$
```

```
$\mathcal{O}^{\mathsf{Prop}}(\mathsf{ID}, \mathsf{gid}, \mathsf{ID}', \mathsf{type})$
1 : require $\mathsf{ST}[\mathsf{ID}] \neq \perp$
2 : $(\gamma, P) \leftarrow\$ \mathsf{prop}(\mathsf{ST}[\mathsf{ID}], \mathsf{gid}, \mathsf{ID}', \mathsf{type})$
3 : reward $P = \perp$ iff $\mathsf{PP}$
4 : if $P = \perp$ return   // failure
5 : reward $(P.\mathsf{ID} \neq \mathsf{ID}') \vee (P.\mathsf{type} \neq \mathsf{type})$
6 : $\mathsf{ViewChecker}(\mathsf{ST}[\mathsf{ID}], \gamma, \mathsf{gid})$
7 : $\mathsf{prop\text{-}ctr} \leftarrow \mathsf{prop\text{-}ctr} + 1$
8 : $\mathsf{T}[\mathsf{gid}, \mathsf{ep}[\mathsf{gid}, \mathsf{ID}], \text{‘prop’}, \mathsf{prop\text{-}ctr}] \leftarrow P$
9 : $\mathsf{ST}[\mathsf{ID}] \leftarrow \gamma$
```

```
$\mathcal{O}^{\mathsf{Create}}(\mathsf{ID}, \mathsf{gid}, G, G^*)$
1 : require $\mathsf{ST}[\mathsf{ID}] \neq \perp$
2 : require $\mathsf{used}[\mathsf{gid}] = \mathsf{false}$   // new gid
3 : $(\gamma, T) \leftarrow\$ \mathsf{create}(\mathsf{ST}[\mathsf{ID}], \mathsf{gid}, G, G^*)$
4 : reward $T = \perp$ iff $\mathsf{OP} \wedge (\mathsf{ID} \in G^* \subseteq G)$
5 : if $T = \perp$ return   // failure
6 : $\mathsf{ViewChecker}(\mathsf{ST}[\mathsf{ID}], \gamma, \mathsf{gid})$
7 : $\mathsf{used}[\mathsf{gid}] \leftarrow \mathsf{true}$
8 : $\mathsf{com\text{-}ctr} \leftarrow \mathsf{com\text{-}ctr} + 1$
9 : $\mathsf{T}[\mathsf{gid}, (-1,-1), \text{‘com’}, \mathsf{com\text{-}ctr}] \leftarrow T$
10 : $\mathsf{ST}[\mathsf{ID}] \leftarrow \gamma$
```

```
$\mathsf{ViewChecker}(\gamma_1, \gamma_2, \mathsf{gid})$
1 : reward $\mathsf{key}(\gamma_1, \mathsf{gid}) \neq \mathsf{key}(\gamma_2, \mathsf{gid})$
2 : reward $\gamma_1[\mathsf{gid}].G \neq \gamma_2[\mathsf{gid}].G$
3 : reward $\gamma_1[\mathsf{gid}].G^* \neq \gamma_2[\mathsf{gid}].G^*$
```

```
$\mathcal{O}^{\mathsf{Commit}}(\mathsf{ID}, \mathsf{gid}, (i_1, \ldots, i_k), \mathsf{com\text{-}type})$
1 : require $\mathsf{ST}[\mathsf{ID}] \neq \perp$
2 : require $\mathsf{ep}[\mathsf{gid}, \mathsf{ID}] \neq (-1,-1)$
3 : $\vec{P} \leftarrow (T[\mathsf{gid}, \mathsf{ep}[\mathsf{gid}, \mathsf{ID}], \mathsf{prop}, i])_{i=(i_1,\ldots,i_k)}$
4 : $(\gamma, T) \leftarrow\$ \mathsf{commit}(\mathsf{ST}[\mathsf{ID}], \mathsf{gid}, \vec{P}, \mathsf{com\text{-}type})$
5 : reward $T = \perp$ iff
        $\mathsf{CP} \wedge (\mathsf{com\text{-}type} \in \mathsf{com\text{-}types})$
6 : if $T = \perp$ return   // failure
7 : reward $\mathsf{ID} \notin \mathsf{ST}[\mathsf{ID}][\mathsf{gid}].G$
8 : $\mathsf{ViewChecker}(\mathsf{ST}[\mathsf{ID}], \gamma, \mathsf{gid})$
9 : $\mathsf{com\text{-}ctr} \leftarrow \mathsf{com\text{-}ctr} + 1$
10 : $\mathsf{T}[\mathsf{gid}, \mathsf{ep}[\mathsf{gid}, \mathsf{ID}], \text{‘com’}, \mathsf{com\text{-}ctr}] \leftarrow T$
11 : $\mathsf{T}[\mathsf{gid}, \mathsf{ep}[\mathsf{gid}, \mathsf{ID}], \text{‘vec’}, \mathsf{com\text{-}ctr}] \leftarrow \vec{P}$
12 : $\mathsf{ST}[\mathsf{ID}] \leftarrow \gamma$
```

```
$\mathcal{O}^{\mathsf{Deliver}}(\mathsf{ID}, \mathsf{gid}, (t, c), c')$
1 : require $\mathsf{ST}[\mathsf{ID}] \neq \perp$
2 : require $\mathsf{ep}[\mathsf{gid}, \mathsf{ID}] \in \{(t,c), (-1,-1)\}$
3 : $T \leftarrow \mathsf{T}[\mathsf{gid}, (t,c), \text{‘com’}, c']$
4 : require $T \neq \perp$
5 : $(\gamma, \mathsf{acc}) \leftarrow \mathsf{proc}(\mathsf{ST}[\mathsf{ID}], T)$
6 : reward $\neg\mathsf{acc}$ iff $\mathsf{DP}$
7 : if $\neg\mathsf{acc}$ return   // failure
8 : reward $\gamma[\mathsf{gid}].G^* \not\subseteq \gamma[\mathsf{gid}].G$
9 : $k \leftarrow \mathsf{key}(\gamma, \mathsf{gid})$
10 : if $\mathsf{ID} \notin \gamma[\mathsf{gid}].G$   // ID removed
11 :   $\mathsf{ep}[\mathsf{gid}, \mathsf{ID}] \leftarrow (-1,-1)$
12 :   reward $k \neq \perp$
13 : else   // ID in group
14 :   $\mathsf{ep}[\mathsf{gid}, \mathsf{ID}] \leftarrow (t+1, c')$
15 :   if $\mathsf{view}[\mathsf{gid}, t+1, c'] = \perp$
16 :     $\mathsf{view}[\mathsf{gid}, t+1, c'] \leftarrow \gamma$
17 :   $\mathsf{ViewChecker}(\mathsf{view}[\mathsf{gid}, t+1, c'], \gamma, \mathsf{gid})$
18 :   if $t \neq -1$   // $T$ not output by create
19 :     $\mathsf{VerChanges}(\mathsf{ST}[\mathsf{ID}], \gamma, T[\mathsf{gid}, (t,c), \text{‘vec’}, c'])$
20 : $\mathsf{ST}[\mathsf{ID}] \leftarrow \gamma$
```

```
$\mathsf{VerChanges}(\gamma_1, \gamma_2, \vec{P})$
1 : $(G_1, G_1^*) \leftarrow (\gamma_1[\mathsf{gid}].G, \gamma_1[\mathsf{gid}].G^*)$
2 : $(G_2, G_2^*) \leftarrow (\gamma_2[\mathsf{gid}].G, \gamma_2[\mathsf{gid}].G^*)$
3 : reward $\mathsf{policy}(\vec{P}, G_1, G_1^*) \neq (G_2, G_2^*)$
```

Figure 4.1: Correctness game for A-CGKA with respect to predicates $(\mathsf{OP}, \mathsf{PP}, \mathsf{CP}, \mathsf{DP})$. Highlighted lines remark the differences between this game and the CGKA correctness game in Figure 3.1.

to CGKA correctness is the protocol invariant $G^* \subseteq G$, which is verified in the $\mathcal{O}^{\mathsf{Create}}$ and $\mathcal{O}^{\mathsf{Deliver}}$ oracles. Besides, all state consistency checks ($\mathsf{ViewChecker}$) and policy verifications ($\mathsf{VerChanges}$) verify the correctness of G^* in addition to the usual G.

The predicates $\mathsf{OP}, \mathsf{PP}, \mathsf{CP}, \mathsf{DP}$ that we introduced for CGKAs could capture additional liveness properties, such that a group member is successfully added to the group of admins whenever an admin commits a proposal of the type $\mathsf{add\text{-}adm}$.

4.2.3 Security

The A-CGKA is a primitive that extends the functionality of a CGKA to provide secure administration mechanisms, but whose main purpose is that the members of a group derive a common group key. Therefore, any A-CGKA construction must achieve the CGKA security notion as presented in Chapter 3, which is based on key indistinguishability. Furthermore, the same considerations as for CGKA apply; there exists a diversity of security models that capture various levels of adversarial activeness, randomness manipulation, sub-optimal cleanness predicates, etc.

Administration security. The main goal of the A-CGKA with respect to a standard CGKA is to prevent that unauthorized (standard) users successfully commit changes in a group. This security notion is different from key indistinguishability, since we do not try to capture the secrecy of the group key, but rather the security of the group evolution. For example, an A-CGKA in which the adversary fully controls a standard group member is not secure with respect to key indistinguishability, but it may still be secure with respect to group evolution if the adversary is not able to inject valid commit messages.

A suitable approach for administration security is to introduce an unpredictability security game (as for correctness in Definition 4.2). The adversary wins if any honest party accepts and successfully processes a commit message which (i) changes the group structure, and (ii) is not sent by a group administrator. Intuitively, this notion resembles signature unforgeability (EUF-CMA) in Appendix A.

Game-based notion. Both key indistinguishability and administration security could be captured in a joint security game. In this case, the cleanness predicate should exclude different sequences of operations for the challenges issued to the group secret, and for the **reward** clauses corresponding to the event of a successful forgery. For instance, a cleanness predicate must exclude the exposure of a standard user and a query to the $\mathcal{O}^{\mathsf{Chall}}$ oracle (following the notation in Figure 3.2) in the same epoch. However, it should not restrict a commit injection attempt in that epoch unless administrative key material is compromised.

Another direction is to capture both properties in different security games. In this case, the adversarial power in the administration security game must be strictly larger than in the CGKA game to ensure that the A-CGKA does not downgrade the security of the CGKA. The definition of A-CGKA security in a game-based model is subject to ongoing work and one of the highest priorities in our project.

Additional notions. There are specific security notions that could be included in an A-CGKA game, depending on the protocol. One possibility is to capture the anonymity of the group administrators with respect to standard members, or with respect to a central server. Another possibility is to capture advanced administration features, such as administrators who have different capabilities, or democratic administrators that require an approval of the commits by more than one admin, as we will discuss in later sections.

4.3 Construction: Individual Admin Signatures

In our first construction, *individual admin signatures* (IAS), each group administrator $\mathsf{ID} \in G^*$ maintains a (unique) signature key pair $(\mathsf{ssk}, \mathsf{spk})$. Such a key pair is independent from the underlying CGKA. Group members keep track of the list of admins G^*, which is provided to them once they join the group and updated after every control message is processed. Every change to the set of admins needs to be shared with all members and signed with a key that belongs to an admin in the list.

4.3.1 Protocol

The IAS construction is presented in Figures 4.2 and 4.3. The first figure describes the A-CGKA algorithms, and the second describes helper functions and auxiliary methods. We note that the algorithms defined in Figure 4.2 are incomplete without the helper functions; therefore, both blocks should be regarded as a single construction.

Primitives. In addition to a CGKA, we make black-box use of three cryptographic primitives (see Appendix A) in our construction:

- **PRGs:** The functions $\{H_k\}_{k\in\mathbb{N}}$ are a family of pseudorandom functions of the form $H_k : R \times \Gamma \to R^k$, where R denotes the randomness space ($R = \{0,1\}^\lambda$) and Γ denotes the state space ($\gamma \in \Gamma$). In practice, we will require functions H_k for $k \leq 5$.
- **KDF:** We require a key derivation function KDF, that we use for one-way hashing of secret values.
- **Signatures:** We make use of a standard signature scheme $\mathcal{S} = (\mathsf{SigGen}, \mathsf{Sig}, \mathsf{Ver})$, where the SigGen algorithm can use explicit randomness if required.

States. As in our correctness game in Figure 4.1, we represent the state of a participant by the symbol γ, which is in turn a dictionary of states, indexed by group identifiers as $\gamma[\mathsf{gid}]$. For each group gid, users keep a separate state that encodes the underlying CGKA state $\gamma[\mathsf{gid}].\mathsf{s0}$, the list of group administrators $\gamma[\mathsf{gid}].\mathsf{adminList}$, and two administration-related signature key pairs $(\gamma[\mathsf{gid}].\mathsf{ssk}, \gamma[\mathsf{gid}].\mathsf{spk})$, $(\gamma[\mathsf{gid}].\mathsf{ssk}', \gamma[\mathsf{gid}].\mathsf{spk}')$. The remaining fields $\gamma[\cdot].\mathsf{ME}$ (representing the local user's ID) and the security parameter $\gamma[\cdot].1^\lambda$ are common to all groups.

The state is initialized by calling the init method, which calls the init method of the underlying CGKA to initialize $\gamma[\mathsf{gid}].\mathsf{s0}$. The remaining fields are set to their corresponding values. All algorithms in our construction, including init, are stateful and do not return an explicit copy of the local state. Instead, they modify the state during runtime. In the event of algorithm failure, the state is not modified and blank values are output.

Randomness. Some of the algorithms that we use throughout the protocol are randomized. In our construction, we make the randomness explicit, including sampled randomness $r_0 \in \{0,1\}^\lambda$ as input. We do so to introduce measures against randomness leakage and manipulation. Before the input randomness r_0 is used in any of the methods, we apply a PRG function $(r_1, \ldots, r_k) \leftarrow H_k(r_0, \gamma)$ that combines the entropy of r_0 and the state γ. Hence, without prior knowledge of γ, an adversary that reads or manipulates r_0 will not be able to derive a corresponding r_i value.

4.3.2 Description

Initialization. Before the creation of a group, a participant starts by calling the init method, which initializes the state γ as described above. $(\mathsf{ssk}, \mathsf{spk})$ and $(\mathsf{ssk}', \mathsf{spk}')$ are two signature key pairs for group administration. The first pair corresponds to signature keys used during protocol execution, while the second pair stores the updated keys after a commit or a key update operation is done by the participant, but before it is processed (i.e. a temporary variable). After successfully processing a commit message, the second key pair replaces the first.

Group creation. The create algorithm creates the group gid from the list of members G, the admin list from G^*, and outputs a control message T for the new members in G. The $\mathsf{adminList}$ variable includes pairs of the form $(\mathsf{ID}, \mathsf{spk}_{\mathsf{ID}})$ for parties $\mathsf{ID} \in G^*$. The signature keys are obtained via the authentication method getSpk except for the group creator, who generates a fresh signature key pair and stores it in the temporary variables $\gamma.\mathsf{ssk}', \gamma.\mathsf{spk}'$. The output of the algorithm is a control message T that is sent to the incoming members, or $\perp$ if the group creation fails (such as when the condition $G^* \subseteq G$ is not satisfied).

```
init($1^\lambda$, ID)
 1 : $\gamma' \leftarrow\$$ CGKA.init($1^\lambda$, ID)
 2 : $\gamma[\cdot]$.s0 $\leftarrow \gamma'$
 3 : $\gamma[\cdot]$.ME $\leftarrow$ ID
 4 : $\gamma[\cdot]$.adminList$[\cdot] \leftarrow \perp$   // stores (ID, spk) pairs
 5 : $\gamma[\cdot]$.ssk, $\gamma[\cdot]$.spk, $\gamma[\cdot]$.ssk$'$, $\gamma[\cdot]$.spk$' \leftarrow \perp$
 6 : $\gamma[\cdot].1^\lambda \leftarrow 1^\lambda$
```

```
create(gid, $G$, $G^*$; $r_0$)
 1 : require ($\gamma$.ME $\in G^*$) $\wedge$ ($G^* \subseteq G$)
 2 : $(r_1, r_2, r_3) \leftarrow H_3(r_0, \gamma)$
 3 : $(W_0, \gamma.s_0) \leftarrow$ CGKA.create(gid, $G$, $\gamma$.s0; $r_1$)
 4 : if $W_0 = \perp$ return $\perp$
 5 : ($\gamma$.ssk$'$, $\gamma$.spk$'$) $\leftarrow$ SigGen($\gamma.1^\lambda$; $r_2$)
 6 : adminList$[\cdot] \leftarrow \perp$
 7 : for ID $\in G^* \setminus \{\gamma$.ME$\}$
 8 :   adminList[ID] $\leftarrow$ ($\gamma$.ME, getSpk(ID, $\gamma$.ME))
 9 : adminList[ME] $\leftarrow$ ($\gamma$.ME, $\gamma$.spk$'$)
10 : $\sigma_W \leftarrow$ Sig($\gamma$.ssk$'$, ($W_0$, adminList); $r_3$)
11 : $T \leftarrow$ (gid, ‘wel’, $\gamma$.ME, $W_0$, adminList, $\sigma_W$)
12 : return $T$
```

```
prop(gid, ID, type; $r_0$)
 1 : $P \leftarrow \perp$;  $(r_1, r_2) \leftarrow H_2(r_0, \gamma)$
 2 : if type = *-adm
 3 :   $P_0 \leftarrow \perp$
 4 :   if type = add-adm
 5 :     $P_0 \leftarrow$ (gid, type, ID, $\gamma$.ME, getSpk(ID, $\gamma$.ME))
 6 :   else if type = rem-adm
 7 :     $P_0 \leftarrow$ (gid, type, ID, $\gamma$.ME, $\perp$)
 8 :   else if type = upd-adm
 9 :     ($\gamma$.ssk$'$, $\gamma$.spk$'$) $\leftarrow$ SigGen($\gamma.1^\lambda$; $r_1$)
10 :     $P_0 \leftarrow$ (gid, type, $\gamma$.ME, $\gamma$.ME, $\gamma$.spk$'$)
11 :   else return $\perp$
12 :   $P \leftarrow (P_0$, Sig($\gamma$.ssk, $P_0$; $r_2$))
13 : else
14 :   ($P$, $\gamma$.s0) $\leftarrow$ CGKA.prop($\gamma$.s0, gid, ID, type; $r_1$)
15 : return $P$
```

```
commit(gid, $\vec{P}$, com-type; $r_0$)
 1 : require com-type $\in$ {adm, std, both}
 2 : $(r_1, \ldots, r_5) \leftarrow H_5(r_0, \gamma)$
 3 : ($\vec{P}_0$, $\vec{P}_A$, admReq) $\leftarrow$ propCleaner($\vec{P}$, gid)
 4 : if admReq $\vee$ (com-type $\in$ {adm, both})
 5 :   require $\gamma$.ME $\in \gamma$.adminList
 6 :   if com-type $\in$ {adm, both}
 7 :     $C_A \leftarrow \vec{P}_A$
 8 :   if com-type $\in$ {std, both}
 9 :     ($C_0$, $W_0$, adminList) $\leftarrow$ c-Std(gid, $\vec{P}_0$, $\vec{P}_A$; $r_3$)
10 :     if $C_0 = \perp$  return $(\perp, \perp)$
      // generate new key pair and sign new spk
11 :   ($\gamma$.ssk$'$, $\gamma$.spk$'$) $\leftarrow$ SigGen($\gamma.1^\lambda$; $r_1$)
12 :   $\sigma_{\mathsf{spk}} \leftarrow$ Sig($\gamma$.ssk, $\gamma$.spk$'$; $r_2$)
13 :   $\sigma_C \leftarrow$ Sig($\gamma$.ssk$'$, ($C_0$, $C_A$); $r_4$)   // could be $\perp$
14 :   $T_C \leftarrow$ (gid, ‘comm’, $\gamma$.ME, $C_0$, $C_A$, $\sigma_C$, $\gamma$.spk$'$, $\sigma_{\mathsf{spk}}$)
15 :   if $W_0 \neq \perp$
        // send updated admins
16 :     $\sigma_W \leftarrow$ Sig($\gamma$.ssk$'$, ($W_0$, adminList); $r_5$)
17 :     $T_W \leftarrow$ (gid, ‘wel’, $\gamma$.ME, $W_0$, adminList, $\sigma_W$)
18 :   else $T_W \leftarrow \perp$
19 : else   // non-admins can commit updates
20 :   ($C_0$, $\perp$, $\perp$) $\leftarrow$ c-Std(gid, $\vec{P}_0$, $\perp$; $r_3$)
21 :   $T_C \leftarrow$ (gid, ‘comm’, $\gamma$.ME, $C_0$, $\perp$, $\perp$, $\perp$, $\perp$)
22 :   $T_W \leftarrow \perp$
23 : return $T_C || T_W$
```

```
proc($T$)
 1 : $(T_C, T_W) \leftarrow T$
 2 : acc $\leftarrow$ false
 3 : if $\gamma$.ME $\notin \gamma$[gid].s0.$G \wedge T_W \neq \perp$
 4 :   require $T_W[1]$ = ’wel’
 5 :   gid $\leftarrow T_W[0]$
 6 :   acc $\leftarrow$ p-Wel($T_W$)
 7 : else if $\gamma$.ME $\in \gamma$[gid].s0.$G \wedge T_C \neq \perp$
 8 :   require $T_C[1]$ = ’comm’
 9 :   gid $\leftarrow T_C[0]$
10 :   acc $\leftarrow$ p-Comm($T_C$)
11 : return acc
```

```
key(gid)
 1 : k $\leftarrow$ CGKA.key($\gamma$.s0, gid)
 2 : return k
```

Figure 4.2: Individual admin signatures (IAS) construction of an A-CGKA, built from a CGKA and a signature scheme $\mathcal{S}$ = (SigGen, Sig, Ver).

```
$\mathsf{updAL}(\mathsf{adminList}, \vec{P}_A)$
---
$\mathbf{for}\ P \in \vec{P}_A$
  $(\bot, \mathsf{type}, \mathsf{ID}, \bot, \mathsf{spk}, \bot) \leftarrow P$
  $\mathbf{if}\ \mathsf{type} \in \{\mathsf{add\text{-}adm}, \mathsf{upd\text{-}adm}\}$
    $\mathsf{adminList}[\mathsf{ID}] \leftarrow (\mathsf{ID}, \mathsf{spk})$
  $\mathbf{if}\ \mathsf{type} = \mathsf{rem\text{-}adm}$
    $\mathsf{adminList}[\mathsf{ID}] \leftarrow \bot$
$\mathbf{return}\ \mathsf{adminList}$
```

```
$\mathsf{propCleaner}(\vec{P}, \mathsf{gid})$
---
$\mathsf{admReq} \leftarrow \mathsf{false}$
$\vec{P}_0, \vec{P}_A \leftarrow []$
$\mathbf{for}\ P \in \vec{P}:$
  $\mathbf{if}\ P[1] = *\text{-}\mathsf{adm}$
    $\mathbf{if}\ \mathsf{VALID}_P$
      $\vec{P}_A \leftarrow [P, \vec{P}_A]$
      $\mathsf{admReq} \leftarrow \mathsf{true}$
  $\mathbf{else}$ // $\vec{P}_0$ is handled by $\mathsf{CGKA}$
    $\vec{P}_0 \leftarrow [P, \vec{P}_0]$
    $\mathbf{if}\ P.\mathsf{type} \in \{\mathsf{add}, \mathsf{rem}\}$
      $\mathsf{admReq} \leftarrow \mathsf{true}$
$\vec{P}_A \leftarrow \mathsf{enforcePolicy}(\vec{P}_A)$
$\mathbf{return}\ (\vec{P}_0, \vec{P}_A, \mathsf{admReq})$
```

```
$\mathsf{c\text{-}Std}(\mathsf{gid}, \vec{P}_0, \vec{P}_A; r_3)$
---
$(C_0, W_0, \gamma.\mathsf{s0}) \leftarrow \mathsf{CGKA.commit}(\mathsf{gid}, \gamma.\mathsf{s0}, \vec{P}_0; r_3)$
$\mathbf{if}\ W_0 \neq \bot$
  $\mathsf{adminList}' \leftarrow \mathsf{updAL}(\gamma.\mathsf{adminList}, \vec{P}_A)$
$\mathbf{return}\ (C_0, W_0, \mathsf{adminList}')$
$\mathbf{else\ return}\ (C_0, \bot, \bot)$
```

```
$\mathsf{VALID}_P$
---
$(\mathsf{gid}, \mathsf{type}, \mathsf{ID}, \mathsf{ID}', \bot, \bot) \leftarrow P$
$S_1 := (P.\mathsf{gid} = \mathsf{gid})$
$S_2 := (\mathsf{ID} \in \gamma[\mathsf{gid}].\mathsf{s0}.G)$
$S_3 := (\mathsf{ID}' \in \gamma[\mathsf{gid}].\mathsf{s0}.G^*)$
$C_1 := (\mathsf{type} = \mathsf{rem\text{-}adm})$
$S_4 := (\mathsf{ID} \in \gamma[\mathsf{gid}].\mathsf{s0}.G^*)$
$C_2 := (\mathsf{type} = \mathsf{add\text{-}adm})$
$\mathbf{return}\ S_1 \wedge S_2 \wedge S_3 \wedge$
    $(\neg C_1 \vee S_4) \wedge (\neg C_2 \vee \neg S_4)$
```

```
$\mathsf{p\text{-}Comm}(T_C)$
---
$(\mathsf{gid}, \bot, \mathsf{ID}, C_0, C_A, \sigma_C, \mathsf{spk}, \sigma_{\mathsf{spk}}) \leftarrow T_C$
$\mathbf{if}\ \sigma_C = \bot$ // no sig $\Rightarrow$ check no changes to $G$
  $(\gamma', \mathsf{acc}) \leftarrow \mathsf{CGKA.proc}(C_0, \gamma[\mathsf{gid}].\mathsf{s0})$
  $\mathbf{if}\ \neg\mathsf{acc} \vee (\gamma'[\mathsf{gid}].\mathsf{s0}.G \neq \gamma[\mathsf{gid}].\mathsf{s0}.G)$
    $\mathbf{return\ false}$
  $\gamma[\mathsf{gid}].\mathsf{s0} \leftarrow \gamma'$
  $\mathbf{return\ true}$
$\mathbf{if}\ \neg[(\mathsf{ID} \in \gamma[\mathsf{gid}].\mathsf{adminList}) \wedge$
    $(\mathsf{Ver}(\mathsf{spk}, (C_0, C_A), \sigma_C)) \wedge$
    $(\mathsf{Ver}(\gamma[\mathsf{gid}].\mathsf{adminList}[\mathsf{ID}].\mathsf{spk}, \mathsf{spk}, \sigma_{\mathsf{spk}}))]$
  $\mathbf{return\ false}$
// check signatures in proposals
$\mathbf{if}\ C_A \neq \bot$
  $\vec{P}_A \leftarrow C_A$
  $\mathbf{for}\ (P, \sigma_P) \in \vec{P}_A:$
    $\mathbf{if}\ \neg\mathsf{Ver}(\mathsf{adminList}[\mathsf{ID}].\mathsf{spk}, P, \sigma_P) \vee P[0] \neq \mathsf{gid}$
      $\mathbf{return\ false}$
// apply commit
$\mathbf{if}\ C_0 \neq \bot$ // proc will succeed
  $(\gamma', \bot) \leftarrow \mathsf{CGKA.proc}(C_0, \gamma.\mathsf{s0})$
  $\mathbf{if}\ \gamma.\mathsf{ME} \notin \gamma'.G$ // user removed
    $\gamma.[\mathsf{gid}] \leftarrow \bot$ // erase state
  $\mathbf{else}\ \gamma[\mathsf{gid}].\mathsf{s0} \leftarrow \gamma'$
// set temporary updated keys
$\mathbf{if}\ (\mathsf{ID} = \gamma.\mathsf{ME}) \vee (\exists P \in C_A : P.\mathsf{ID}' = \gamma.\mathsf{ME})$
  $(\gamma.\mathsf{ssk}, \gamma.\mathsf{spk}) \leftarrow (\gamma.\mathsf{ssk}', \gamma.\mathsf{spk}')$
  $\gamma.\mathsf{ssk}', \gamma.\mathsf{spk}' \leftarrow \bot$
$\gamma.\mathsf{adminList} \leftarrow \mathsf{updAL}(\gamma.\mathsf{adminList}, \vec{P}_A)$
$\gamma.\mathsf{adminList}[\mathsf{ID}].\mathsf{spk} \leftarrow \mathsf{spk}$
$\mathbf{return\ true}$
```

```
$\mathsf{p\text{-}Wel}(T_W)$
---
$(\mathsf{gid}, \bot, \mathsf{ID}, W_0, \mathsf{adminList}, \sigma_W) \leftarrow T_W$
$M \leftarrow (W_0, \mathsf{adminList})$
$\mathbf{if}\ \mathsf{Ver}(\mathsf{adminList}[\mathsf{ID}].\mathsf{spk}, M, \sigma_{W_0}) = 0$
  $\mathbf{return\ false}$
$(\gamma', \mathsf{acc}) \leftarrow \mathsf{CGKA.proc}(W_0)$
$\mathbf{if}\ \mathsf{acc}$
  $\gamma[\mathsf{gid}].\mathsf{s0} \leftarrow \gamma'$
  $\gamma[\mathsf{gid}].\mathsf{adminList}[\mathsf{ID}] \leftarrow \mathsf{adminList}$
$\mathbf{return}\ \mathsf{acc}$
```

Figure 4.3: Helper functions for the IAS construction in Figure 4.2.

Proposals. The prop algorithm creates a proposal of the input type by calling the corresponding CGKA.prop algorithm, without introducing modifications in the group. We note that any user (depending on the CGKA construction, sometimes even a non-member) can craft a standard proposal message; however, only admins will be able to create an administrative proposal message as such a proposal is signed. The signature is included to prevent an (insider) adversary from forging the sender of the proposal in an attempt to impersonate an admin. A proposal creation does not have any effect on the state other than the storage of temporary keys in case of an update.

In the case of an add-adm proposal to elevate ID to an admin, the proposer $\gamma.\mathsf{ME}$ retrieves a public signature key spk from ID using a generic function getSpk. Such authentication mechanism is not modelled in our construction and we leave it for future work. Nevertheless, the CGKA may implement or require some authentication mechanism for handling add proposals; one possibility is to implement getSpk in the same way.

Commits. The commit algorithm can only be called by group administrators (except for the special case in which only key updates are proposed, when standard users can commit), and performs the following actions:

1. Clean the input vector of proposals $\vec{P}$, ensuring that they are well-formed. This is done via the propCleaner algorithm, which in turn calls the enforcePolicy method. We do not define an explicit policy in this construction, but one could enforce, for instance, the MLS policy (removing duplicates, prioritizing removals, etc.) [BBM+]. In addition, we ensure the validity of the admin proposals by checking the predicate VALID_P. This predicate verifies that the gid matches, that added users (respectively admins) do not belong to G respectively G^*), that removed users do belong to G (respectively G^*), and that the proposer is an admin.

2. Carry out the administrative and the standard commits and produce an administrative commit message C_A, a standard CGKA commit C_0, and an updated adminList. For this purpose, the algorithm calls the helper functions c-Adm and c-Std.

3. Update the administrative signature key ssk' and sign it using the old signature key ssk.

4. Produce the final control message T. The message T is divided in two components[1]. A first component includes all the required information for existing ACGKA members, such as the administrative updates. This is named

[1]This division is made for clarity, but is also an efficiency improvement: the welcome part of a control message does not need to be sent to existing group members, and the commit part can be sent to previous members only.

T_W (standing for welcome), following notation in works such as [ACDT20, KPPW+21]. A second component T_C (for commit) is intended for existing members. In case the vector of proposals does not require changes to the group structure or to the group administration (i.e., only updates), standard users are allowed to perform a commit.

Processing control messages. The proc method takes a control message T as input and updates the state accordingly. The algorithm returns an acceptance bit acc which is true if the processing succeeds, in which case the state is updated. Otherwise, the state remains the same (i.e., the method provides robustness). During the proc execution, several checks are required before the state is updated. For newly added users, p-Wel verifies the message signature on the adminList, attempts to process the message via the underlying CGKA, and updates the state given this succeeds. For existing members, p-Com verifies the administrator signature and the signatures in the admin proposals. The state is updated if all verification succeeds; a removed user blanks their state, and temporary keys are updated if necessary.

Key retrieval. The key method simply outputs the main CGKA group secret corresponding to the current user's state (i.e. to the current epoch).

4.3.3 Features

Design. Despite the length of the construction, we believe that the construction is, at least conceptually, as simple as an A-CGKA construction can get. Some of the redundancy comes from modifying only temporary variables until every important steps in the algorithm have suceeded; this is made to guarantee robustness as we argued in Section 2.3.2.

The IAS protocol can be built on top of any CGKA, while the only additional primitive that is required is a signature scheme. In fact, signatures are often already present in CGKAs, such as in [AJM20]; in these cases, the extension from CGKA to A-CGKA can be rather direct.

Commit and propose policies. Our construction allows standard users to perform a commit if there are no changes in the group structure or in the administration. This feature is an optional design choice that should not affect security (and could be reflected in a correctness predicate). It is possible to construct an A-CGKA that does not allow standard users to commit at all.

On the other hand, we do not allow standard users to propose administrative changes (even if these could be later ignored by admins). Doing so would require an additional protection mechanism for proposals, since an (insider or active) adversary could trivially forge some of the proposal fields. For example, the proposer's ID' could be replaced by an administrator's to impersonate an admin.

Security mechanisms. The security in the group administration is provided by the admin signatures; an adversary should not be able to commit changes to the group unless it compromises the state of one of the group administrations. These signatures can be updated by the administrators at any point, and their update (which introduces new randomness) is indeed required whenever a new commit is carried out. Hence, the administration mechanism is forward secure and also post-compromise secure.

The simplicity of the adminList comes at the price of having to store and update it locally for every group member, which increases communication complexity, since every signature update has to be broadcast to the group. Besides, administrative actions are undeniable and traceable (there is no possibility to hide which admin is performing an action) both by group members and by a central server. Separately, achieving security guarantees for incoming users is not straightforward; additional protections are needed to ensure that the list of group administrators is not forged.

As we discuss in later sections, security can be strengthened with the introduction of threshold signatures [Sho00], in which a number $m \geq 1$ of group administrators would be required to sign a commit. However, this involves a substantial overhead. A future work direction is the introduction of *message framing* as in [AJM20], which consists of encrypting and MACing every control message using the epoch key. This feature improves metadata protection and mitigates the impact of network adversaries that can alter control messages (for example, by cutting some parts of the message or by mixing it with other messages).

4.4 Construction: Dynamic Group Signature

In our second construction, *dynamic group signature* (DGS), the group administrators agree on a common, unique signature key pair that they use for signing administrative messages. To agree on a secret, they run a separate CGKA. As opposed to IAS, now group administrators can be opaque to group members (as long as the CGKA's maintain certain anonymity properties), since the only identifier for admins is the admin signature, which is shared. Besides, group members do not need to keep track of an administrator list.

The protocol shares many characteristics with IAS, such as the general structure, notation, and even some lines of code. Hence, our exposition is less descriptive and often refers to the previous section.

4.4.1 Protocol

The DGS construction is presented in Figures 4.4 and 4.5. As before, the first figure describes the A-CGKA algorithms, and the second describes helper functions and auxiliary methods. During the algorithm, we refer to the main (or standard) CGKA as CGKA, and to the administrative CGKA as CGKA^*. The purpose of the first is that group members agree on a common secret, whereas the second serves only for administrative purposes (i.e., admins deriving a common signature key).

We require the use of the same primitives as before, besides an additional CGKA. We note that CGKA and CGKA^* are not necessarily the same protocol - indeed, such a property can be used by a protocol designer if, for instance, stronger FS and PCS is required for the administrative CGKA. Derandomization is also done in an analogous way, by using the family $\{H_k\}_k$ of PRGs.

States. Now, the state for a single group $\gamma[\mathsf{gid}]$ includes two separate state variables: $\gamma[\mathsf{gid}].\mathsf{s0}$ corresponding to the primary CGKA, and $\gamma[\mathsf{gid}].\mathsf{sA}$ corresponding to CGKA^*. Besides, the state includes a field for the administrative public key $\gamma[\mathsf{gid}].\mathsf{spk}$, which is known by all group members to enable verification. Since we require to keep two separate CGKA states $\mathsf{s0}$, sA, which in practice correspond to two different groups, we also need to have two independent (but related) group identifiers. We simply write gid for the primary CGKA and gid^* for the admin CGKA. One can generate such a pair by taking a unique identifier s and appending $\mathsf{gid} = s||0$ and $\mathsf{gid}^* = s||1$.

4.4.2 Description

Initialization. The init procedure initializes the state γ and has to be run by every party before executing any other algorithm.

Group creation. The create algorithm creates two separate CGKAs: the standard CGKA and the administrative CGKA^*, by calling the corresponding CGKA create methods. These output new states $\mathsf{s0}$ and sA, which are stored; and control messages W_0 and W_A, which are collated into a *create* control message $T = T_{CR}$.

Proposals. The prop algorithm checks the type of the proposal to be created and generates a proposal message P accordingly. No checks on the user are carried out; in case $\gamma.\mathsf{ME}$ is not in the group, the proposal will simply fail if the CGKA or CGKA^* policies do not allow external users to propose changes. As before, authentication is not modelled and we assume that the problem is handled in each of the CGKAs.

Commits. As in IAS, commit can be called by group administrators, and also by standard users if there are no changes to the group structure. The steps followed by the algorithm are also very similar. The main differences are two. First, there is

```
init(1^λ, ID)
1 : $\gamma[\cdot]$.s0 ← CGKA.init($1^\lambda$, ID)
2 : $\gamma[\cdot]$.sA ← CGKA*.init($1^\lambda$, ID)
3 : $\gamma[\cdot]$.ME ← ID
4 : $\gamma[\cdot]$.spk ← $\bot$
5 : $\gamma[\cdot].1^\lambda \leftarrow 1^\lambda$

create(gid, $G, G^*; r_0$)
1 : require ($\gamma$.ME $\in G^*$) $\wedge$ ($G^* \subseteq G$)
2 : $(r_1, r_2) \leftarrow H_2(r, \gamma)$
3 : $(W_0, \gamma$.s0) ← CGKA.create(gid, $G, \gamma$.s0; $r_1$)
4 : $(W_A, \gamma$.sA) ← CGKA*.create(gid*, $G^*, \gamma$.sA; $r_2$)
5 : $T_{CR}$ ← (gid, 'create', $W_0, W_A$)
6 : return $T_{CR}$

proc($T$)
1 : $(T_{CR}, T_W, T_C) \leftarrow T$
2 : if $T_{CR} \neq \bot$
3 :    require $T_{CR}[1]$ = 'create'
4 :    if $\gamma$.ME $\in \gamma$[gid].s0.$G$   return false
5 :    gid ← $T_{CR}[0]$
6 :    $(\bot, \bot, W_0, W_A) \leftarrow T$
7 :    (acc$_1$, $\gamma$[gid].s0) ← CGKA.proc($W_0$)
8 :    (acc$_2$, $\gamma$[gid].sA) ← CGKA*.proc($W_A$)
9 :    acc ← acc$_1 \wedge$ acc$_2$   // use robustness
10 : else if $\gamma$.ME $\notin \gamma$[gid].s0.$G \wedge T_W \neq \bot$
11 :    require $T_W[1]$ = 'wel'
12 :    gid ← $T_W[0]$
13 :    acc ← p-Wel($T_W$)
14 : else if $\gamma$.ME $\in \gamma$[gid].s0.$G \wedge T_C \neq \bot$
15 :    require $T_C[1]$ = 'comm'
16 :    gid ← $T_W[0]$
17 :    acc ← p-Comm($T_C$)
18 : return acc

prop(gid, ID, type; $r_0$)
1 : $r \leftarrow H_1(r_0, \gamma)$
2 : if type ∈ {add-adm, rem-adm, upd-adm}
3 :    ($\gamma$[gid].s0, $P$) ← CGKA*.prop($\gamma$.sA, gid*, ID, type; $r$)
4 : else
5 :    ($\gamma$[gid].sA, $P$) ← CGKA.prop($\gamma$.s0, gid, ID, type; $r$)
6 : return $P$

commit(gid, $\vec{P}$, com-type; $r_0$)
1 : require com-type ∈ {adm, std, both}
2 : $(r_1, \ldots, r_5) \leftarrow H_5(r_0, \gamma)$
3 : $C_0, \sigma_{C_0}, C_A, W_A, \sigma_{\mathsf{spk}}, W_0, \sigma_{W_0} \leftarrow \bot$
4 : $(\vec{P}_0, \vec{P}_A$, admReq) ← propCleaner($\vec{P}$, gid)
5 : if admReq ∨ (com-type ∈ {adm, both})
6 :    require $\gamma$.ME $\in \gamma$.sA.$G$
7 :    (ssk, spk) ← getSigKey(CGKA*.key(gid, $\gamma$.sA))
8 :    if type ∈ {adm, both}
9 :       (ssk, spk, $\sigma_{\mathsf{spk}}, C_A, W_A$) ←
              c-Adm(gid, $\vec{P}_A$, ssk; $r_1, r_2$)
10 :    if type ∈ {std, both}
11 :       $(C_0, \sigma_{C_0}, W_0, \sigma_{W_0})$ ←
              c-Std(gid, $\vec{P}_0$, ssk; $r_3, r_4, r_5$)
12 :    $T_C$ ← (gid, 'comm', $C_0, \sigma_{C_0}, C_A, W_A$, spk, $\sigma_{\mathsf{spk}}$)
13 :    $T_W$ ← (gid, 'wel', $W_0, \sigma_{W_0}$, spk)
14 : else
15 :    $(C_0, \bot, \bot, \bot)$ ← c-Std(gid, $\vec{P}_0, \bot; r_3, r_4, r_5$)
16 :    $T_C$ ← (gid, 'comm', $C_0, \bot, \bot, \bot, \bot, \bot$)
17 :    $T_W \leftarrow \bot$
18 : return $T_C || T_W$

key(gid)
1 : k ← CGKA.key($\gamma$.s0, gid)
2 : return k
```

Figure 4.4: Dynamic group signature (DGS) construction of an A-CGKA, built from two (possibly different) CGKAs, CGKA and CGKA*, and a signature scheme $\mathcal{S}$ = (SigGen, Sig, Ver).

$\text{c-Adm}(\text{gid}, \vec{P}_A, \text{ssk}_{\text{prev}}; r_1, r_2)$

```
for $P \in \vec{P}_A$
  if $P.\text{type} = \text{add-adm}$
    require $P.\text{ID} \in \gamma.\text{s0}.G$
$(C_A, W_A, \gamma.\text{sA}) \leftarrow$
        $\text{CGKA}^*.\text{commit}(\text{gid}^*, \gamma.\text{sA}, \vec{P}_A; r_1)$
if $C_A = \perp$  return $\perp$
// state processed temporarily
$st_{\text{temp}} \leftarrow \text{CGKA}^*.\text{proc}(C_A)$
$(k, \perp) \leftarrow \text{CGKA}^*.\text{key}(\text{gid}^*, st_{\text{temp}})$
$(\text{ssk}, \text{spk}) \leftarrow \text{getSigKey}(k)$
$\sigma_{\text{spk}} = \text{Sig}(\text{ssk}_{\text{prev}}, \text{spk}; r_2)$
return $(\text{ssk}, \text{spk}, \sigma_{\text{spk}}, C_A, W_A)$
```

$\text{c-Std}(\text{gid}, \vec{P}_0, \text{ssk}; r_3, r_4, r_5)$

```
$(C_0, W_0, \gamma.\text{s0}) \leftarrow$
        $\text{CGKA}.\text{commit}(\text{gid}, \gamma.\text{s0}, \vec{P}_0; r_3)$
if $C_0 = \perp$  return $\perp$
$\sigma_{C_0}, \sigma_{W_0} \leftarrow \perp$
if $\text{ssk} \neq \perp$
  $\sigma_{C_0} \leftarrow \text{Sig}(\text{ssk}, C_0; r_4)$
    if $W_0 \neq \perp$
    $\sigma_{W_0} \leftarrow \text{Sig}(\text{ssk}, W_0; r_5)$
return $(C_0, \sigma_{C_0}, W_0, \sigma_{W_0})$
```

$\text{propCleaner}(\vec{P}, \text{gid})$

```
$\text{admReq} = \text{false}$
$\vec{P}_0, \vec{P}_A \leftarrow []$
for $P \in \vec{P}$ :
  if $P[0] \neq \text{gid}$ break
  if $P[1] = *\text{-adm}$
    $\vec{P}_A \leftarrow [P, \vec{P}_A]$
    $\text{admReq} = \text{true}$
  else
    $\vec{P}_0 \leftarrow [P, \vec{P}_0]$
    if $P.\text{type} \in \{\text{add}, \text{rem}\}$
      $\text{admReq} = \text{true}$
    // admin rem from G $\Longrightarrow$ rem from G*
    if $(P.\text{type} = \text{rem}) \wedge (P.\text{ID} \in \gamma.\text{sA}.G)$
      $P \leftarrow \text{CGKA}^*.\text{prop}(\gamma.\text{sA}, \text{gid}^*, P.\text{ID}, \text{rem}; \perp)$
    $\vec{P}_A \leftarrow [\vec{P}_A, P]$
$(\vec{P}_0, \vec{P}_A) \leftarrow \text{enforcePolicy}(\vec{P}_0, \vec{P}_A)$
return $(\vec{P}_0, \vec{P}_A, \text{admReq})$
```

$\text{p-Comm}(T_C)$

```
$(\text{gid}, \perp, C_0, \sigma_{C_0}, C_A, W_A, \text{spk}, \sigma_{\text{spk}}) \leftarrow T_C$
$\gamma' \leftarrow \gamma.\text{sA}$
if $\sigma_{C_0} = \perp$   // no sig $\Rightarrow$ check no changes to $G$
  $(\gamma', \text{acc}) \leftarrow \text{CGKA}.\text{proc}(C_0, \gamma[\text{gid}].\text{s0})$
  if $\neg\text{acc} \vee (\gamma'[\text{gid}].\text{s0}.G \neq \gamma[\text{gid}].\text{s0}.G)$
    return false
  $\gamma[\text{gid}].\text{s0} \leftarrow \gamma'$
  return true
else if $\neg(\text{Ver}(\text{spk}, C_0, \sigma_{C_0}) \wedge \text{Ver}(\gamma.\text{spk}, \text{spk}, \sigma_{\text{spk}}))$
  return false
if $\gamma.\text{ME} \in \gamma.\text{sA}.G$
  $(\gamma', \text{acc}) \leftarrow \text{CGKA}^*.\text{proc}(C_A, \gamma.\text{sA})$
else if $W_A \neq \perp$
  $(\gamma', \text{acc}) \leftarrow \text{CGKA}^*.\text{proc}(W_A, \gamma.\text{sA})$
  if $\neg\text{acc}$  return false
if $C_0 \neq \perp$
  $(\gamma^\dagger, \text{acc}^\dagger) \leftarrow \text{CGKA}.\text{proc}(C_0, \gamma.\text{s0})$
  if $\neg\text{acc}^\dagger$  return false
  $\gamma[\text{gid}].\text{s0} \leftarrow \gamma^\dagger$
  if $\gamma.\text{ME} \neq \gamma^\dagger.G$   // removed user
    $\gamma[\text{gid}] \leftarrow \perp$
    return true
$\gamma[gid].\text{sA} \leftarrow \gamma'$; $\gamma[\text{gid}].\text{spk} \leftarrow \text{spk}$
return true
```

$\text{p-Wel}(T_W)$

```
$(\text{gid}, \perp, W_0, \sigma_{W_0}, \text{spk}) \leftarrow W$
require $\gamma.\text{ME} \notin \gamma[\text{gid}].\text{s0}.G$
if $\neg\text{Ver}(\text{spk}, W_0, \sigma_{W_0})$  return false
$(\gamma', \text{acc}) \leftarrow \text{CGKA}.\text{proc}(W_0, \gamma.\text{s0})$
if $\neg\text{acc}$  return false
$\gamma[\text{gid}].\text{s0} \leftarrow \gamma'$
$\gamma[\text{gid}].\text{spk} \leftarrow \text{spk}$
return true
```

$\text{getSigKey}(r)$

```
$(\text{ssk}, \text{spk}) \leftarrow \text{SigGen}(1^\lambda; H_1(r))$
return $(\text{ssk}, \text{spk})$   // deterministic in r
```

Figure 4.5: Helper functions for the DGS construction in Figure 4.4.

no adminList to be updated. Second, the old signature key pair (ssk, spk) is derived from the key of CGKA*, and a new signature key pair is created in c-Adm after the update of the CGKA*. In order to obtain the future group secret, c-Adm requires to process its own admin commit C_A temporarily. In case of a standard commit, where the administrative CGKA is not updated, ssk is not evolved.

A commit outputs two control messages which, again, are collated into a unique message: A welcome message T_W for newly added users, and a commit message T_C for group members. In the latter, we also include the welcome messages to the administrative CGKA, since they must always be addressed to current group members. We note that commits in both CGKAs are totally independent: the primary CGKA can be updated while the administrative CGKA is not, and vice-versa.

Processing control messages. The proc method takes a control message T, determines the type of message (create, welcome, or commit), updates the state accordingly, and returns an acceptance bit acc. The robustness property is maintained. For newly added users, p-Wel verifies the admin signature, processes the welcome message using CGKA.proc and stores the new public admin key spk. For existing members, p-Comm verifies the administrator signature (unless the commit does not require administrative rights). Then, depending on the commit type, the primary CGKA, the administrative CGKA, or both CGKAs are updated. In case $T = T_{CR}$ is a create message, both CGKAs process their respective welcome messages separately.

Key retrieval. The key method outputs the current epoch key, corresponding to the key in the primary CGKA. The administrative signature key pair can be obtained by group admins by calling the method getSigKey, with the CGKA* key as input.

4.4.3 Features

Some parts of the discussion provided for the IAS protocol apply to DGS as well. In particular, the security mechanisms are also based on signatures, we follow similar commit and propose policies, we aim to include message framing, and we maintain robustness throughout our protocol. Besides, our DGS protocol presents some unique features that we introduce below.

Minimal reveal of information. As opposed to IAS, the set of group administrators is opaque to both the central server and to the rest of the group (whenever the underlying CGKAs preserve the anonymity of the members with respect to external parties). In case this protocol is implemented using a central server, the information given to the server is reduced further. Besides, the verification issues for newly added users are mitigated. The administrative spk can be a public value that a server can store, since it does not provide any relevant information about the group. Hence, an incoming member can verify the authenticity of an administrative signature by retrieving spk directly from the server, or from a different channel other than the welcome message itself. One possibility is out-of-bound verification of safety numbers, which is a feature provided in two-party chats but not in group chats in messaging apps such as WhatsApp.

Double CGKAs. A noticeable feature from the DGS protocol is its modularity. The protocol allows the use of two distinct and independent CGKA protocols. This makes our construction very flexible since it is possible to precise the exact level of security that goes into the group administration. For example, in a scenario where active attacks are especially relevant, one could utilize heavy but robust CGKA protocols such as the P-Act-Rob in [ACJM20], which uses NIZKs to prevent state forks. We believe that such flexibility can be very useful in decentralized constructions, such as the DCGKA in [WKHB20]. On the other hand, running a CGKA among group administrators requires additional resources from the admins, and increases communication complexity (although only for the subset of admins, since the rest of the group is opaque to it).

We remark the simplicity of achieving PCS and FS in the group administration keys (in the adversarial model for the CGKA^*), since both properties are ensured by the CGKA^* itself. Delegation and revocation of admin keys are also straightforward.

4.5 Discussion

In this section, we provide a final analysis of the A-CGKA primitive and the IAS and DGS constructions, including a security discussion and a comparison with respect to other possible approaches to group administration. We also discuss possibilities for extending our work.

4.5.1 Security

Due to the lack of a security model for A-CGKA, we cannot formally prove the security of our protocols; we defer this task to future work. Nonetheless, following the administration security approach introduced in Section 4.2.3, we are able to provide some informal insights.

Unforgeability. In both constructions, the administrative functionality is guaranteed by digital signatures. The goal is that an adversary that manages to forge a correct admin-restricted commit message needs to either (i) have access to administrative key material, or (ii) forge the signature on the control message. Hence, in a future security proof, the notion of signature unforgeability (see Appendix A) will play a central role.

Key material updates. Both of our protocols are designed to provide PCS and FS to the administrative key material, in the same way as a CGKA protocol does for the group key. Every time a new admin signature key pair $(\mathsf{ssk}, \mathsf{spk})$ is generated, spk is signed using the previous secret signature key. The signature ensures the authenticity of the freshly generated signature while providing forward security. If the randomness used in the signature key generation is fresh, PCS is also achieved.

Randomness manipulation. The randomness that is used throughout every local instance of the protocol is made explicit and combined with the user's state. This mitigates the effect of randomness manipulation, since if the adversary is able to manipulate the randomness used in the protocol, there is still some entropy available to the user in its state. Hence, the user's state is not compromised. Nevertheless, randomness manipulation still prevents the self-healing effects of an update, meaning that a party needs to sample fresh randomness to recover from a state compromise.

4.5.2 Efficiency

Group administrators may improve the efficiency of messaging protocols as less members are allowed to perform changes in the group structure. This can be seen for example in the Tainted TreeKEM protocol [KPPW+21], where there are noticeable efficiency improvements with respect to standard TreeKEM if only a small fraction of the members are allowed to commit.

Regarding the efficiency of our own constructions, in IAS administrative commits and updates have to be delivered to and processed by every group member. This is opposite to DGS, in which the group admins remain opaque to the other group members and do not need to receive the administration messages; this reduces communication complexity.

4.5.3 Alternatives

We evaluated alternative A-CGKA solutions, other than IAS and DGS. An alternative to DGS is to use *ring signatures* [RST01] to sign commits made by administrators. Ring signatures are non-interactive signature schemes for groups of users (the administrators), in which a user signs on behalf of a group of users, but the anonymity of the signer is preserved. For PCS and FS new ring signature pair should be created for every commit. Their main drawback is the involved overhead, both in efficiency and in implementation since ring signatures are not considered a standard primitive.

We also considered *admin credentials*, which are administration certificates that can be issued to any group member by the group creator. Every time a commit is made, the certified admin can attach the credential to the commit message so that users can verify the administration signature. The method presents issues regarding admin removals, since the group creator (or another admin) must broadcast the revocation of the credential to the whole group. Besides, it is not clear how to achieve FS and PCS for the signature in the credential, and if any administrator besides the group creator is allowed to issue credentials, it seems hard to avoid including chains of signatures.

4.5.4 Advanced Administration

Our model of cryptographic administrators captures the essential properties of group administrators: a total control of the list of members of the group, revocation of administrative power, and designation of new group administrators. Nevertheless, one can conceive more advanced settings in which the group admins have additional capabilities or security properties.

Admin hierarchies. In some messaging apps such as Telegram and WhatsApp, the group creator has stronger capabilities than other admins. For instance, the group creator can never be removed from the group by another admin. In this sense, one can conceive a hierarchy of administrators of several levels, such as $G^{**} \subseteq G^{*} \subseteq G$ where G^{**} are super-administrators. The administration can even be fine-grained, in which different admins have different sets of permissions.

Democratic administrators. One of the main issues of our A-CGKA constructions is that security breaks completely if the state of an admin is compromised or if an admin deviates from the protocol. To improve on the robustness of the protocol, democratic administrators can be built. A group with democratic administrators would require $t \geq 1$ votes, by t different group administrators, to validate a commit. In the IAS construction, one could simply require t valid signatures on the commit message. Other possibility is the use of threshold signatures, which must be signed by t users to be valid [Sho00].

Participant muting. Another real-world functionality provided in some messaging apps is muting members of a group. It is possible to enforce this property using cryptography. One approach is that the group administrators run a secondary A-CGKA protocol in which the common group secret is used to derive a signature key pair $(\mathsf{ssk}, \mathsf{spk})$. To send a message, members must sign their messages using spk. If a participant is muted, it suffices to remove them from the secondary A-CGKA. Then, muted members will be able to filter messages from other muted users (since they could still be informed of spk), but they will not able to sign their own. We note that the filtering can also be done by a central server who discards all messages who have a missing or invalid signature, without revealing any additional information to the server.

If anonymity is a concern, there exist solutions such as Opaque Vetted Encryption [HS20], which maintains the sender anonymous from the server and from an external observer, but allows the server to filter messages from vetted (muted) users. Nevertheless, we note that in messaging services where the identity of the group members is known, muted members can generally bypass a ban by sending individual messages to all group members using two-party messages.

Chapter 5

Conclusions and Future Work

5.1 Conclusions

The development of provably secure group messaging protocols presents important challenges. In the messaging literature, there exists a relatively common framework for modelling two-party protocols that allows for direct comparisons of their security models. In group messaging, however, the literature is limited and the considered approaches present major differences. A large amount of work is still required in order to lay down stable foundations for the construction of protocols.

In this thesis, we started by providing several insights in group messaging, including a novel definition of CGKA correctness, an analysis of TreeKEM variants, and a discussion on the main properties of a group messaging protocol. Our main contribution is a formal study of the administration of a messaging group. We presented two separate constructions of cryptographic group administrators: individual admin signatures (IAS), constructed on top of a CGKA by introducing signature key pairs associated to each group administrator; and dynamic group signature (DGS), constructed using two independent CGKA protocols and a common group signature key pair. To formalize a natural framework for our constructions, we introduced the A-CGKA abstraction, which is a cryptographic primitive that extends CGKA to capture group administration, including a correctness definition and some insights on the desired security.

Both our constructions and our A-CGKA formalism serve as a basis for future work in group messaging. The introduction of administrators in a CGKA enhances security in several aspects, particularly when insider adversaries are modelled. Furthermore, we believe that our ideas can also be applied towards the construction of decentralized messaging protocols, in which concurrency becomes a significant issue for secure group evolution.

5.2 Limitations and Current Directions

Our results on group administration present important limitations; many of them due to the large scope of the project; there is ongoing work to address these issues. Our definition of A-CGKA is limited since we do not specify an adversarial model and a definition of security. Also, our two constructions have not been proven correct under our own correctness definition. Currently, these directions have the highest priority in our work. Other issues to be addressed in the near future are the following:

- Compare the security of our constructions to other security models, and study whether our constructions could potentially diminish the security of an underlying CGKA protocol (for example, in the active adversarial setting).
- Consider *message framing* as in MLS [BBM$^+$, AJM20] to improve security against active adversaries and malicious insiders.
- Address the problem of authentication, studying different possibilities to authenticate administrators and standard users.
- Analyze multi-group security, especially in the case where long-term identity keys are used by the underlying CGKA.
- Formalize protocol composition in A-CGKAs, especially in our DGS construction where two CGKAs are combined.

On a different note, our work has been mostly theoretical and we have neither implemented our protocols nor studied their efficiency in practical scenarios, such as when they are implemented on top of existing protocols such as TreeKEM. A practical analysis would be a valuable complement for this thesis.

5.3 Future Work

Besides our current directions, we lay down several research lines that we consider relevant, both related to group administration and to group messaging in general.

- Explore advanced administration properties, and how to extend this to other applications besides group messaging. Fine-grained administration, where each admin has a different set of permissions, is also an interesting avenue. One possibility is to use of attribute-based encryption or matchmaking encryption [AFNV19], but these incur nontrivial overheads.
- Construct and prove secure group messaging protocols, possibly with group administrators, that use different forms of authentication than PKIs. For example, password-based authentication and out-of-bound authentication. This is an important step in the design of decentralization-friendly protocols.

- Analyze different ways of reducing trust on the central server, eventually leading to fully decentralized protocols (including authentication mechanisms) which are efficient and secure, extending the work in [WKHB20]. One possibility is to make their model more realistic by achieving consensus among the group administrators, solving concurrency issues.

- Study how the techniques used in non-ratcheted group key agreement protocols, such as those surveyed in [PRSS21], can be applied to CGKAs. In particular, study the feasibility of extending their protections against active adversaries to ratcheted protocols that ensure post-compromise security.

- Other possible avenues in which there exists room for improvement are: handling concurrency, improving efficiency in existing protocols (for example, reducing the communication complexity of protocols like TreeKEM by avoiding broadcasting), formalizing real-world schemes (such as the Sender Keys protocol), including deniability in group chats, and capturing multi-device security.

Bibliography

[ACD19] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the signal protocol. In *EUROCRYPT*, volume 11476 LNCS, pages 129–158. Springer Verlag, 2019. 1, 7, 11, 12, 14, 17, 20, 78

[ACDT20] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the ietf mls standard for group messaging. In *CRYPTO*, volume 12170 LNCS, pages 248–277. Springer, 2020. 1, 3, 5, 21, 23, 24, 25, 26, 30, 32, 33, 34, 35, 37, 38, 39, 54

[ACJM20] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In *Theory of Cryptography Conference*, volume 12551 LNCS, pages 261–290. Springer, 2020. 3, 21, 23, 24, 30, 33, 41, 60, 79

[AFNV19] Giuseppe Ateniese, Danilo Francati, David Nuñez, and Daniele Venturi. Match me if you can: Matchmaking encryption and its applications. In *CRYPTO*, volume 11693 LNCS, pages 701–731. Springer, 2019. 66

[AJM20] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the Insider Security of MLS. *IACR Cryptol. ePrint Arch.*, 2020:1327, 2020. 4, 21, 23, 24, 30, 33, 35, 39, 40, 41, 45, 54, 55, 66, 79

[BBM$^+$] R Barnes, B Beurdouche, J Millican, E Omara, K Gohn-Gordon, and R Robert. The Messaging Layer Security (MLS) Protocol. 20, 26, 33, 34, 35, 53, 66

[BBR18] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups A protocol proposal for Messaging Layer Security (MLS). https://hal.inria.fr/hal-02425247, 2018. 5, 20, 23, 33

[BCK21] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Cryptographic Security of the MLS RFC, Draft 11. *IACR Cryptol. ePrint Arch.*, 2021:137, 2021. 20

[BDR20] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. In *Theory of Cryptography Conference*, volume 12551 LNCS, pages 198–228. Springer, 2020. 4, 22

[BGB04] 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*, pages 77–84, 2004. 2, 3, 7, 13

[BK17] Muhammad Bilal and Shin-Gak Kang. A secure key agreement protocol for dynamic group. *Cluster Computing*, 20(3):2779–2792, 2017. 22

[BR93] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In *Proceedings of the 1st ACM Conference on Computer and Communications Security*, pages 62–73, 1993. 80

[BRV20] Fatih Balli, Paul Rösler, and Serge Vaudenay. Determining the core primitive for optimally secure ratcheting. In *ASIACRYPT*, volume 12493 LNCS, pages 621–650. Springer, 2020. 16

[BSJ$^+$17] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In *CRYPTO*, volume 10403 LNCS, pages 619–650. Springer, 2017. 13, 29

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *Proceedings 42nd IEEE Symposium on Foundations of Computer Science*, pages 136–145. IEEE, 2001. 30, 41, 80

[CDV21] Andrea Caforio, F Betül Durak, and Serge Vaudenay. Beyond Security and Efficiency: On-Demand Ratcheting with Security Awareness. In *Public Key Cryptography (2)*, pages 649–677, 2021. 13, 15, 17

[CGCD$^+$20] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the signal messaging protocol. *Journal of Cryptology*, 33, 2020. 11

[CGCG16] Katriel Cohn-Gordon, Cas Cremers, and Luke Garratt. On post-compromise security. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 164–178. IEEE, 2016. 2

[CGCG$^+$18] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In *Proceedings of the 2018*

ACM SIGSAC Conference on Computer and Communications Security, pages 1802–1819, 2018. 20, 33

[CHK21] Cas Cremers, Britta Hale, and Konrad Kohbrok. The Complexities of Healing in Secure Group Messaging: Why Cross-Group Effects Matter. In *30th {USENIX} Security Symposium ({USENIX} Security 21)*, pages 1847–1864, 2021. 23, 24

[DV19] F Betül Durak and Serge Vaudenay. Bidirectional Asynchronous Ratcheted Key Agreement with Linear Complexity. In *International Workshop on Security*, pages 343–362. Springer, 2019. 1, 7, 13, 14, 15, 16, 17

[Gaj20] Phillip Gajland. On Asynchronous Group Key Agreements: Tripartite Asynchronous Ratchet Trees, 2020. 10, 12, 33

[GS02] Craig Gentry and Alice Silverberg. Hierarchical id-based cryptography. In *ASIACRYPT*, pages 548–566. Springer, 2002. 16

[HL02] Jeremy Horwitz and Ben Lynn. Toward hierarchical identity-based encryption. In *EUROCRYPT*, pages 466–481. Springer, 2002. 16

[HS20] Martha Norberg Hovd and Martijn Stam. Vetted encryption. In *INDOCRYPY*, volume 12578 LNCS, pages 488–507. Springer, 2020. 62

[HW20] Andreas Hülsing and Florian Weber. Epochal Signatures for Deniable Group Chats. *IACR Cryptol. ePrint Arch.*, 2020:1138, 2020. 3, 23

[JMM19] Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-Optimal Guarantees for Secure Messaging. In *EUROCRYPT*, volume 11476 LNCS, pages 159–188. Springer, 2019. 1, 13, 17

[JS18] Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In *CRYPTO*, volume 10991 LNCS, pages 33–62. Springer Verlag, 2018. 7, 13, 14, 16

[KPPW$^+$21] K. Klein, G. Pascual-Perez, M. Walter, C. Kamath, M. Capretto, M. Cueto, I. Markov, M. Yeo, J. Alwen, and K. Pietrzak. Keep the Dirt: Tainted TreeKEM, Adaptively and Actively Secure Continuous Group Key Agreement. In *2021 IEEE Symposium on Security and Privacy (SP)*, pages 268–284, Los Alamitos, CA, USA, 2021. IEEE Computer Society. 3, 21, 23, 24, 25, 26, 30, 33, 39, 40, 44, 54, 61

[KS05] Jonathan Katz and Ji Sun Shin. Modeling insider attacks on group key-exchange protocols. In *Proceedings of the 12th ACM conference on Computer and communications security*, pages 180–189, 2005. 4, 33, 40, 45

[LVH13] 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*, pages 249–254, 2013. 23

[Mar] Moxie Marlinspike. Signal Protocol, GitHub Repository. https://github.com/signalapp/libsignal-protocol-java/tree/master/java/src/main/java/org/whispersystems/libsignal. 2, 3, 19

[OBR$^+$] E Omara, B Beurdouche, E Rescorla, S Inguva, A Kwon, and A Duric. The Messaging Layer Security (MLS) Architecture. 20, 21

[PM16] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. https://signal.org/docs/specifications/doubleratchet/, 2016. 2, 7, 9, 10, 12

[PR18] Bertram Poettering and Paul Rösler. Asynchronous ratcheted key exchange. Technical report, Cryptology ePrint Archive, Report 2018/296, 2018. 7, 13, 14, 15, 16

[PRSS21] Bertram Poettering, Paul Rösler, Jörg Schwenk, and Douglas Stebila. Sok: Game-based security models for group key exchange. In *Topics in Cryptology–CT-RSA 2021: Cryptographers' Track at the RSA Conference 2021, Virtual Event, May 17–20, 2021, Proceedings*, page 148. Springer Nature, 2021. 20, 22, 23, 24, 26, 29, 33, 40, 67

[RMS18] Paul Rosler, Christian Mainka, and Jorg Schwenk. More is Less: On the End-to-End Security of Group Chats in Signal, WhatsApp, and Threema. In *3rd IEEE European Symposium on Security and Privacy, EURO S&P 2018*, pages 415–429. Institute of Electrical and Electronics Engineers Inc., 2018. 4, 44, 45

[RST01] Ronald L Rivest, Adi Shamir, and Yael Tauman. How to leak a secret. In *ASIACRYPT*, pages 552–565. Springer, 2001. 61

[Sho00] Victor Shoup. Practical threshold signatures. In *EUROCRYPT*, pages 207–220. Springer, 2000. 55, 62

[Tel] Telegram. Group Chats on Telegram. https://telegram.org/tour/groups. 3, 44

[UDB$^+$15] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: secure messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249. IEEE, 2015. 19

[Wei19] Matthew A Weidner. Group Messaging for Secure Asynchronous Collaboration, 2019. 19, 21, 33

[Wha16] Whatsapp encryption overview, 2016. 3, 20, 43

[WKHB20] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. *IACR Cryptol. ePrint Arch.*, 2020:1281, 2020. 3, 21, 45, 60, 67

[YV20] Hailun Yan and Serge Vaudenay. Symmetric asynchronous ratcheted communication with associated data. In *IWSEC 2020: Advances in Information and Computer Security*, volume 12231 LNCS, pages 184–204. Springer, 2020. 17

Appendix A

Cryptography Background

A.1 Cryptographic Primitives

We introduce several standard cryptographic primitives that are used or mentioned throughout the thesis. We will often use these as black-boxes. In the following definitions, $\lambda \in \mathbb{N}$ is the security parameter, which is typically related to the key size. We say that a function $f(\lambda) = \mathsf{negl}(\lambda)$ if for every constant $c > 0$. there exists a $\lambda_0 > 0$ such that $f(\lambda) < 1/\lambda^c$ for every $\lambda > \lambda_0$.

Definition A.1 (Collision resistant hash function). *A family of functions* $\{f_\lambda\}_\lambda$ *such that* $f_\lambda : \{0,1\}^{n(\lambda)} \to \{0,1\}^{m(\lambda)}$ *is collision resistant if for any polynomial time adversary* $\mathcal{A}$*,*

$$\Pr[(x_1, x_2) \leftarrow \mathcal{A}(\lambda) : x_1 \neq x_2 \wedge f_\lambda(x_1) = f_\lambda(x_2)] = \mathsf{negl}(\lambda).$$

Definition A.2 (Symmetric encryption). *A symmetric or private-key cryptosystem is a pair of algorithms* $(\mathsf{Enc}, \mathsf{Dec})$ *such that:*

- $c \leftarrow \mathsf{Enc}(\mathsf{k}, m)$ *encrypts a plaintext* m*.*
- $m' \leftarrow \mathsf{Dec}(\mathsf{k}, c)$ *decrypts a ciphertext* c*.*

We say that the cryptosystem is correct if $\forall m \in \mathcal{P}, \forall \mathsf{k} \in \mathcal{K}$*, then*

$$\mathsf{Dec}(\mathsf{k}, \mathsf{Enc}(\mathsf{k}, m)) = m$$

where $\mathcal{K}$ *is the set of possible keys and* $\mathcal{P}$ *is the set of possible plaintexts.*

Definition A.3 (Public-key encryption). *A public-key cryptosystem is a triple of algorithms* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *such that:*

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\!\$\ \mathsf{Gen}(1^\lambda)$ *creates a pair of public and private keys. If the randomness* r *involved in the generation is known, we can write* $(\mathsf{sk}, \mathsf{pk}) \leftarrow \mathsf{Gen}(1^\lambda; r)$*.*

- $c \leftarrow\!\$\, \mathsf{Enc}(\mathsf{pk}, m)$ *encrypts a plaintext* m.
- $m' \leftarrow \mathsf{Dec}(\mathsf{sk}, c)$ *decrypts a ciphertext* c.

We say that the cryptosystem is correct when for any λ*, any* $m \in \mathcal{P}$*, and any randomness, if* $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\$\, \mathsf{Gen}(1^\lambda)$*, then*

$$\mathsf{Dec}(\mathsf{sk}, \mathsf{Enc}(\mathsf{pk}, m)) = m.$$

Definition A.4 (KEM)**.** *A key encapsulation mechanism (KEM) is a triple of algorithms* $(\mathsf{Gen}, \mathsf{Encaps}, \mathsf{Decaps})$ *such that:*

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\$\, \mathsf{Gen}(1^\lambda)$ *creates a public-private key pair.*
- $(\mathsf{k}, c) \leftarrow\!\$\, \mathsf{Encaps}(\mathsf{pk})$ *creates an encapsulated key* k *and a ciphertext* c.
- $\mathsf{k} \leftarrow \mathsf{Decaps}(\mathsf{sk}, c)$ *recovers the key* k.

We say that the KEM is correct when, for any λ *and any randomness, if* $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\$\, \mathsf{Gen}(1^\lambda)$ *and* $(\mathsf{k}, c) \leftarrow\!\$\, \mathsf{Encaps}(\mathsf{pk})$*, we have that*

$$\mathsf{Decaps}(\mathsf{sk}, c) = \mathsf{k}.$$

We note that a KEM can be built from any public-key cryptosystem in a straightforward manner. The Gen algorithm is the same, the encapsulation selects a random $\mathsf{k} \leftarrow\!\$\, \{0,1\}^\lambda$ and sets $c = \mathsf{Enc}(\mathsf{pk}, \mathsf{k})$, and the decapsulation is done as $\mathsf{k} = \mathsf{Dec}(\mathsf{sk}, c)$. In the other direction, we can build a public-key cryptosystem using a KEM and a symmetric cryptosystem; we simply need to generate a key using $\mathsf{Encaps}(\mathsf{pk})$ and encrypt the plaintexts m under such key.

Definition A.5 (MAC)**.** *A message authentication code (MAC) is a pair of algorithms* $(\mathsf{Tag}, \mathsf{Ver})$ *such that:*

- $t \leftarrow \mathsf{Tag}(\mathsf{k}, m)$ *outputs a tag* t *from a key* k *and a message* m.
- $b \leftarrow \mathsf{Ver}(\mathsf{k}, t, m)$ *outputs* $b \in \{0,1\}$*, indicating acceptance or rejection, given a key* k*, a tag* t *and a message* m.

We say that the MAC is correct if for any $\mathsf{k} \in \mathcal{K}$ *and* $m \in \mathcal{P}$*, where* $\mathcal{K}$ *is the set of valid keys, then*

$$\mathsf{Ver}(\mathsf{k}, \mathsf{Tag}(\mathsf{k}, m), m) = 1.$$

Definition A.6 (AEAD)**.** *An authenticated encryption with associated data is a pair of algorithms* $(\mathsf{Enc}, \mathsf{Dec})$ *such that:*

- $c \leftarrow \mathsf{Enc}(\mathsf{k}, m, \mathsf{ad})$ *encrypts a plaintext* m *given a key* k *and associated data* ad.
- $m' \leftarrow \mathsf{Dec}(\mathsf{k}, c, \mathsf{ad})$ *decrypts a ciphertext* c *using a key* k *and associated data* ad.

We say that the cryptosystem is correct if for all valid pairs $(\mathsf{k}, \mathsf{ad})$ *and for any* $m \in \mathcal{P}$*, we have*

$$\mathsf{Dec}(\mathsf{k}, \mathsf{Enc}(\mathsf{k}, m.\mathsf{ad}), \mathsf{ad}) = m.$$

Definition A.7 (Digital signature)**.** *A digital signature scheme is a triple of algorithms* $(\mathsf{SigGen}, \mathsf{Sig}, \mathsf{Ver})$ *such that:*

- $(\mathsf{sk}, \mathsf{pk}) \leftarrow\$ \mathsf{SigGen}(1^\lambda)$ *creates a public-private key pair.*
- $\sigma \leftarrow\$ \mathsf{Sig}(\mathsf{sk}, m)$ *generates a signature* σ *from a message* m *and the secret key* sk*.*
- $b \leftarrow \mathsf{Ver}(\mathsf{pk}, \sigma, m)$ *outputs* $b \in \{0, 1\}$*, indicating acceptance or rejection, given a signature* σ*, a message* m *and a public key* pk*.*

We say that the signature scheme is correct when for any λ*, any* $m \in \mathcal{P}$*, and any randomness, if* $(\mathsf{sk}, \mathsf{pk}) \leftarrow\$ \mathsf{SigGen}(1^\lambda)$ *and* $\sigma \leftarrow\$ \mathsf{Sig}(\mathsf{sk}, m)$*, then*

$$\mathsf{Ver}(\mathsf{pk}, \sigma, m) = 1.$$

Definition A.8 (KDF)**.** *A key derivation function is an algorithm that derives one or several cryptographic keys from one or several secret seeds. More precisely,* $(\mathsf{k}_1, \ldots, \mathsf{k}_m) \leftarrow \mathsf{KDF}(s_1, \ldots, s_n)$*.*

Definition A.9 (HIBE)**.** *A hierarchical identity based encryption scheme consists of four algorithms* $(\mathsf{Setup}, \mathsf{Del}, \mathsf{Enc}, \mathsf{Dec})$ *defined as follows:*

- $(\mathsf{pp}, \mathsf{k}) \leftarrow\$ \mathsf{Setup}(1^\lambda)$ *generates public parameters* pp *and an identity key* k *for the master identity vector* $\Delta = []$*.*
- $\mathsf{k}' \leftarrow\$ \mathsf{Del}(\mathsf{k}, \Delta, \Gamma)$ *delegates a key* k' *for the identity* $[\Delta, \Gamma]$ *given the key* k*, identity vector* Δ *and the suffix* Γ*.*
- $c \leftarrow\$ \mathsf{Enc}(\mathsf{pp}, \Delta, t, m)$ *produces a ciphertext* c *given public parameters* pp*, an identity vector* Δ*, a tag* t*, and a message* m*.*
- $m \leftarrow \mathsf{Dec}(\mathsf{k}, t, c)$ *outputs a message* m *given a key* k*, a tag* t *and a ciphertext* c*.*

Every key k *is associated to an identity* Δ *and obtained using the* Setup *or* Del *algorithms. We say that a HIBE is correct if every message* m *encrypted to identity* Δ *can be decrypted using a key* k *associated to the identity* Δ*.*

A.2 Capturing Security

In this thesis, security is captured by games, following standard practice in the literature. A security game consists of a sequence of operations executed between a challenger and an adversary $\mathcal{A}$, following certain rules. The challenger sets up the game, and then the adversary is allowed to make calls to a set of oracles, determined by the security notion to be captured. We will deal with two main types of games as described in [ACD19]:

- **Unpredictability:** The goal of $\mathcal{A}$ is to cause a specific game-winning event T, which is captured by a win clause. For example, if $\mathcal{A}$ manages to forge a signature. The advantage $\mathsf{Adv}^{\mathsf{unp}}_{\mathcal{A}}(\lambda)$ of the adversary in the game is given by

$$\mathsf{Adv}^{\mathsf{unp}}_{\mathcal{A}}(\lambda) = \Pr[T = 1].$$

- **Indistinguishability:** There are two games IND_b parametrized by a bit $b \in \{0,1\}$ that the challenger can play with $\mathcal{A}$. The goal of $\mathcal{A}$ is to distinguish which of the two games is being played, i.e., output a guess b' for b. For example, in game $b = 0$, $\mathcal{A}$ is given real ciphertexts, as opposed to random strings if $b = 1$. The advantage $\mathsf{Adv}^{\mathsf{ind}}_{\mathcal{A}}(\lambda)$ of the adversary in the game is given by

$$\mathsf{Adv}^{\mathsf{ind}}_{\mathcal{A}}(\lambda) = \left|\Pr[\mathsf{IND}^{\mathcal{A}}_0(1^\lambda) = 1] - \Pr[\mathsf{IND}^{\mathcal{A}}_1(1^\lambda) = 1]\right|.$$

Depending on the type of adversary, we can consider *adaptive* or *non-adaptive* security. Broadly speaking, a scheme is secure against an adaptive adversary when, during a security game, the result of a query can influence future queries by the adversary (i.e., the game is interactive). In contrast, a non-adaptive adversary must announce all its queries in advance. We introduce a series of security notions that are used throughout the thesis, starting with standard security notions for public-key cryptosystems.

Definition A.10 (IND-CPA security). *A public-key cryptosystem* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is secure against chosen plaintext attacks (IND-CPA) if for any polynomial-time adversary* $\mathcal{A}$*, the advantage* $\mathsf{Adv}^{\text{ind-cpa}}_{\mathcal{A}}(\lambda) = \mathsf{negl}(\lambda)$ *in the following game:*

```
IND-CPA$^{\mathcal{A}}_{b,\mathsf{Enc}}(\lambda)$
1 :  $(\mathsf{sk}, \mathsf{pk}) \leftarrow\!\$\ \mathsf{Gen}(1^\lambda)$
2 :  $(\mathsf{st}, m_0, m_1) \leftarrow\!\$\ \mathcal{A}(1^\lambda, \mathsf{pk})$
3 :  $c \leftarrow\!\$\ \mathsf{Enc}(\mathsf{pk}, m_b)$
4 :  $b' \leftarrow\!\$\ \mathcal{A}(1^\lambda, \mathsf{pk}, c, \mathsf{st})$
5 :  return $b'$
```

Definition A.11 (IND-CCA security). *A public-key cryptosystem* $(\mathsf{Gen}, \mathsf{Enc}, \mathsf{Dec})$ *is secure against chosen ciphertext attacks (IND-CCA) if for any polynomial-time adversary* $\mathcal{A}$*, the advantage* $\mathsf{Adv}^{\text{ind-cpa}}_{\mathcal{A}}(\lambda) = \mathsf{negl}(\lambda)$ *in the following game:*

```
$\text{IND-CCA}^{\mathcal{A}}_{b,\mathsf{Enc}}(\lambda)$
1 : $(\mathsf{sk}, \mathsf{pk}) \leftarrow\$ \mathsf{Gen}(1^\lambda)$
2 : $(\mathsf{st}, m_0, m_1) \leftarrow\$ \mathcal{A}^{\mathcal{O}^{\mathsf{Dec1}}}(1^\lambda, \mathsf{pk})$
3 : $c_0 \leftarrow\$ \mathsf{Enc}(\mathsf{pk}, m_b)$
4 : $b' \leftarrow\$ \mathcal{A}^{\mathcal{O}^{\mathsf{Dec2}}}(1^\lambda, \mathsf{pk}, c_0, \mathsf{st})$
5 : return $b'$
```

```
$\mathcal{O}^{\mathsf{Dec1}}(c)$
1 : return $\mathsf{Dec}(\mathsf{sk}, c)$

$\mathcal{O}^{\mathsf{Dec2}}(c)$
1 : require $c \neq c_0$
2 : return $\mathsf{Dec}(\mathsf{sk}, c)$
```

Notice that any IND-CPA/CCA secure cryptosystem must be randomized and yield different encryptions of the same plaintext. Otherwise, the adversary could just query the decryption oracle on encryptions of m_0 and m_1 and then compare the resulting ciphertexts.

The security of signatures is captured by unforgeability games, in which the goal of the adversary is to craft a valid signature without the knowledge of the private key. The most standard security notion is EUF-CMA security.

Definition A.12 (EUF-CMA security)**.** *A signature scheme* $(\mathsf{SigGen}, \mathsf{Sig}, \mathsf{Ver})$ *is existentially unforgeable under chosen message attacks (EUF-CMA) if for any polynomial-time adversary* $\mathcal{A}$*, the advantage* $\mathsf{Adv}^{\text{euf-cma}}_{\mathcal{A}}(\lambda) = \mathsf{negl}(\lambda)$ *in the following game:*

```
$\text{EUF-CMA}^{\mathcal{A}}_{\mathsf{Sig}}(\lambda)$
1 : $(\mathsf{sk}, \mathsf{pk}) \leftarrow\$ \mathsf{SigGen}(1^\lambda)$
2 : $\mathcal{L} \leftarrow \emptyset$
3 : $(m_0, \sigma) \leftarrow\$ \mathcal{A}^{\mathcal{O}^{\mathsf{Sig}}}(1^\lambda, \mathsf{pk})$
4 : require $m_0 \notin \mathcal{L}$
5 : reward $\mathsf{Ver}(\mathsf{pk}, \sigma, m_0) = 1$
```

```
$\mathcal{O}^{\mathsf{Sig}}(m)$
1 : $\mathcal{L} \leftarrow \mathcal{L} \cup \{m\}$
2 : return $\mathsf{Sig}(\mathsf{sk}, m)$
```

A.3 Security Models

There are mainly two security models that we consider in this thesis: the Standard Model (SM) and the Random Oracle Model (ROM). We also mention the Universal Composability (UC) framework, used in some group messaging works in the literature [ACJM20, AJM20].

Standard Model A proof of security in the SM means that the security proof relies only on standard cryptographic assumptions, which are specified. For example, the DDH (Decisional Diffie-Hellman) assumption or the RSA assumption. In the SM, the only assumption allowed for hash functions is that they are collision-resistant and preimage-resistant.

Random Oracle Model The ROM was introduced by Bellare and Rogaway [BR93] and has since then been used to formalize the security of many cryptographic protocols. Nowadays, proofs in the ROM are relatively standard. In the ROM, hash functions are modelled as random oracles. A random oracle is a function that, on input x, returns a uniformly random string if the input has not been queried before, but it is consistent with previous answers during the whole security game. More specifically, a random oracle $H(x)$ is defined as:

```
$H(x)$ :
1 : if $x \in \mathcal{L}$
2 :   return $\mathsf{dict}[x]$
3 : else
4 :   $\mathcal{L} \leftarrow \mathcal{L} \cup \{x\}$
5 :   $r \leftarrow\$ \{0,1\}^{\mathsf{poly}(\lambda)}$
6 :   $\mathsf{dict}[x] \leftarrow r$
7 :   return $r$
```

Random oracles are a heuristic. In fact, there exist constructions secure in the ROM in which *any* instantiation of the random oracle as a hash function is insecure. Hence, proving security in the ROM is not as satisfactory as proving it in the Standard Model.

Universal Composability The UC framework was devised by Canetti in [Can01] and has since then evolved notably. The framework can be used to prove the security of cryptographic protocols with respect to a composition operation, called *universal composition*. Protocols that are proven secure in the UC remain secure in the presence of an arbitrary number of concurrent and adversarially controlled protocol sessions. Hence, the UC framework is suitable to model scenarios such as tasks running in an operating system and protocols with multiple instances.

The proofs in the UC model require a real world and an ideal world. The real world corresponds to the protocol π in question. In the ideal world, an ideal functionality $\mathcal{F}$ that characterizes the security and correctness of π is defined. The environment $\mathcal{Z}$, which is adversarially controlled, controls the inputs and the outputs of concurrent local instances of π, and interacts with $\mathcal{F}$ and with a simulator $\mathcal{S}$ that replicates the adversary in the ideal world. The goal is to prove that the output of π in the environment $\mathcal{Z}$ (real world) and the output of $\mathcal{F}$ (ideal world) are indistinguishable.