On the Efficiency and Security of Secure Group Messaging

by

Guillermo Pascual-Pérez

September, 2024

A thesis submitted to the
Graduate School
of the
Institute of Science and Technology Austria
in partial fulfillment of the requirements
for the degree of
Doctor of Philosophy

Committee in charge:

Beatriz Vicoso, Chair
Krzysztof Pietrzak
Joël Alwen
Alon Rosen

Institute of
Science and
Technology
Austria

The thesis of Guillermo Pascual-Pérez, titled *On the Efficiency and Security of Secure Group Messaging*, is approved by:

Supervisor: Krzysztof Pietrzak, ISTA, Klosterneuburg, Austria

Signature: _____

Committee Member: Joël Alwen, AWS Wickr

Signature: _____

Committee Member: Alon Rosen, Bocconi University, Milan, Italy

Signature: _____

Defense Chair: Beatriz Vicoso, ISTA, Klosterneuburg, Austria

Signature: _____

Signed page is on file

ISTA Thesis, ISSN: 2663-337X

I hereby declare that this thesis is my own work and that it does not contain other people's work without this being so stated; this thesis does not contain my previous work without this being stated, and the bibliography contains all the literature that I used in writing the dissertation.

I declare that this is a true copy of my thesis, including any final revisions, as approved by my thesis committee, and that this thesis has not been submitted for a higher degree to any other university or institution.

I certify that any republication of materials presented in this thesis has been approved by the relevant publishers and co-authors.

Signature: _____

Guillermo Pascual-Pérez
September, 2024

Signed page is on file

Abstract

Instant messaging applications like Whatsapp, Signal or Telegram have become ubiquitous in today's society. Many of them provide not only end-to-end encryption, but also security guarantees even when the key material gets compromised. These are achieved through frequent key update performed by users. In particular, the compromise of a group key should preserve confidentiality of previously exchanged messages (*forward secrecy*), and a subsequent key update will ensure security for future ones (*post-compromise security*). Though great protocols for one-on-one communication have been known for some time, the design of ones that scale *efficiently* for larger groups while achieving akin security guarantees is a hard problem. A great deal of research has been aimed at this topic, much of it under the umbrella of the Messaging Layer Security (MLS) working group at the IETF. Started in 2018, this joint effort by academics and industry culminated in 2023 with the publication of the first standard for secure group messaging [BBR+23].

At the core of secure group messaging is a cryptographic primitive termed *Continuous Group Key Agreement*, or CGKA [ACDT21], that essentially allows a changing group of users to agree on a common key with the added functionality security against compromises is achieved by users asynchronously issuing a key update. In this thesis we contribute to the understanding of CGKA across different angles. First, we present a new technique to effect dynamic operations in groups, i.e., add or remove members, that can be more efficient that the one employed by MLS in certain settings. Considering the setting of users belonging to multiple overlapping groups, we then show lowerbounds on the communication cost of constructions that leverage said overlap, at the same time showing protocols that are asymptotically optimal and efficient for practical settings, respectively. Along the way, we show that the communication cost of key updates in MLS is average-cost optimal. An important feature in CGKA protocols, particularly for big groups, is the possibility of executing several group operations concurrently. While later versions of MLS support this, they do at the cost of worsening the communication efficiency of future group operations. In this thesis we introduce two new protocols that permit concurrency without any negative effect on efficiency. Our protocols circumvent previously existing lower bounds by satisfying a new notion of post-compromise security

that only asks for security to be re-established after a certain number of key updates have taken place. While this can be slower than MLS in terms of rounds of communication, we show that it leads to more efficient overall communication. Additionally, we introduce a new technique that allows group members to decrease the information they need to store and download, which makes one of our protocols enjoy much lower download cost than any other existing CGKA constructions.

Acknowledgements

As I grow older, I increasingly realize of the importance that external influences and stimuli have on one's mood, opinions, ideas and, in fact, in almost every aspect of one's life. Milestones like completing a PhD are no different, and indeed it would be incredibly naïve to think that I would have gotten here without the support and inspiration of a huge number of people out there (some of which I am probably forgetting). It is a delight to be able to thank them here: this section is possibly the most rewarding to write.

I would like to start by thanking my supervisor Krzysztof Pietrzak for his continued support and for creating a pressure free working environment with lots of creative freedom. I am also grateful to the rest of my PhD defense committee: Alon Rosen, and Joël Alwen, as well as Beatriz Vicoso, for kindly agreeing to chair. Joël deserves a special mention as he has been an insatiable source of problems and ideas, and of valuable advice. I am lucky to have been part of a research group that shares much more than discussions about cryptography. Hamza, Chethan, Karen, Michael, Michelle, Ahad, Miguel, Benedikt, Charlotte, Christoph, Suvradip, Akin and Anshu, it has been a pleasure. They have offered guidance and a helping hand when needed, and have been excellent company in conferences, trips and other outings. A special thanks goes to Ben, who has made work so much more enjoyable with his humor and patience, and has been my steady companion in almost every research venture, and even beyond in our shared woodworking shop.

I would also like to thank PQShield and, in particular, Thomas Prest, for welcoming me in Paris and for insightful discussions, and Shuichi Katsumata, who has been a wonderful collaborator and whose excitement for new ideas is admirable. My gratefulness also goes to the rest of my co-authors Keitaro Hashimoto, Marta Mularczyk and Yiannis Tselekounis. To Martin Albrecht, for welcoming me in London and for inspiring discussions about cryptography, its social foundations, and woodworking. Also to Cory Myers and Giulio Berra at the Freedom of the Press Foundation, for their initial trust and their tireless patience and humbleness during our discussions. Further, deep gratitude goes to the people in the wider cryptography community who make it feel so welcoming, particularly to the wonderful David Balbás and Paul Rösler. I am also

grateful to the SICT community, and especially Sophia, for giving me new hope and perspectives about our role and responsibilities as technologists.

I would also like to thank those that supported my professional journey before arriving to ISTA. In particular, Christophe Petit and Giacomo Micheli for getting me interested in cryptography in the first place during a summer project back in my undergraduate years, and for the real attention and patience they showed me. Finally, I am grateful to Zubin, who was the most excellent tutor I could have wished for through university, and who continued to be a reliable source of support afterwards.

Now, it would be even more naïve to think that I am here only thanks to people that belong to my work sphere. I am indescribably lucky to have people around me that not only support me, but make my human experience so rich and worthwhile. Even though the work presented here is not the subject of most (if any) of our conversations – and will likely not be read by (m)any –, the journey to write this thesis would lack meaning without them. They all, whether part of my present or my past, have my sincerest love.

First and foremost, I would like to thank my parents, for their pride in me, for getting me here, for believing in me and having worked and cared more for me than anyone will ever do. Despite the physical distance that separates us, I hope we can keep feeling close to each other.

I would like to thank all the people at ISTA that made my settling in here so much more smooth and enjoyable: Mims, Michelle, Djordje, Dario, Mariano, Sarath, Martin, Lizzie, Patricia, Sebastiano and, especially, Nataliia, for all of her love and caring support.

I am beyond grateful to everyone that contributed to making Vienna feel cozy and exciting and, above all, human. Beginning with my housemates Lara, Nadine and Matzi, who make me feel so at home and so cared for. To Rebecca, Raquel and Jaime, for our adventures in the world of crafting and creativity and for enabling me to keep my connection with my spanish identity in Vienna. To Ruth, for our beautiful conversations and friendship, and for her coherence and being an endless source of inspiration. To Manu, for sharing my passion for wood, for allowing me to see her and for seeing me, in all of our weirdness and beauty. To Sven, for his kindness and sweetness. To Charlie, for her honesty and inspiring transparency. To Lukas, for his honest and deep care, towards me and everyone around him. To Juro and Maggie, for teaching me the value of slowness and being present. To the XwhY community, for showing me spaces of softness and vulnerability. To Bella, for her beautiful and supportive way of communication. To Franzi, Ruby, Lydia, Martin, Mareike, Lea, Jana, Lisa, Kerstin, Mila, Kathi, Rosalie, Christian, Camille, Lukas, Consi, Schaki and all the others who have gifted me fond memories and with whom I have shared moments of joy and difficulty. And to Auri, my most fervent supporter, for her deepest warmth and love, for teaching about the joy of living and the power of loving and trusting oneself. I learn from you

every day.

Finally, I am also grateful to those friends that are far away. To Kat, for our adventures together, for being a source of inspiration and glee, and for a friendship that, even after 6 years living apart, still feels like home. To Aneeka, for her continued care of our friendship through my unreliability when using the very motivation for this thesis: messaging apps. To Nere, for her honesty and vulnerability, and for a friendship that still grows after being apart for so many years. To Tom, for his humor and daring trust when joining me in wild adventures. To Alex, Vero and Leo, for being so selfless and inspiring, and for making possible that we share joy and dreams in the most beautiful rural community: Frondeira. And to everyone that I could thus share this joy and dreams with: Beltrán, Raúl A., Miguel, Raúl G., Marco, Cheska, Lorenzo, Angie, Emilie, Emma... To Jesus, Maria, Ale, Leire, Cris, Cesar and the many others that make going back to Madrid a joyful experience. And last, but most definitely not least, to Javi and Lu, thanks to whom I never feel alone. Javi for knowing me better than myself and for being my most consistent and loving source of support, kindness and honesty throughout the years. Lu for her contagious illusion for life, her ability to get people together, and her deep love, curiosity and humbleness. They both inspire me, and I have the luxury and pride of walking this life with them.

Funding. This work was funded by the European Union's Horizon 2020 research and innovation programme under the Marie Skłodowska-Curie Grant Agreement No. 665385.

About the Author

Guillermo Pascual-Perez obtained a Master of Mathematics (MMATH) degree from the University of Oxford, specializing in pure mathematics before deciding to jump to cryptography and joining ISTA in September 2018. He is interested in the interplay between theory and practice in cryptography, and the impact that it has in society. His work, thanks in no small part to his able collaborators, has been published at premier venues in cryptography (EC, TCC, CT-RSA, PKC, SCN) and security (S&P) communities. Outside of his research activities, he can be found compensating his intellectual work with hands-on hobbies like gardening and woodworking.

List of Collaborators and Publications

1. Chapter 3 is based on "Karen Klein[†], Guillermo Pascual-Perez[†], Michael Walter[†], Chethan Kamath, Margarita Capretto, Miguel Cueto, Ilia Markov, Michelle Yeo, Joël Alwen, Krzysztof Pietrzak. *Keep the Dirt: Tainted TreeKEM, Adaptively and Actively Secure Continuous Group Key Agreement*. In 42nd IEEE Symposium on Security and Privacy, **SP 2021** [KPPW+21]".

   [significant contributions to the protocol design and efficiency analysis]

2. Chapter 4 is based on "Joël Alwen, Benedikt Auerbach, Mirza Ahad Baig, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, Michael Walter[‡] *Grafting Key Trees: Efficient Key Management for Overlapping Groups*. In Theory of Cryptography **TCC 2021** [AAB+21b]".

   [significant contributions to the protocols design, minor contributions to the lower bounds proofs]

3. Chapter 5 is based on "Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzyzstof Pietrzak, Michael Walter[‡]. *CoCoA: Concurrent Continouous Group Key Agreement*. In Annual International Conference on the Theory and Applications of Cryptographic Techniques - **EUROCRYPT 2022**[AAN+22b]".

   [main responsible for the security proof; significant contributions to the protocol design, main responsible for the partial states technique]

4. Chapter 6 is based on "Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak[‡]. *DeCAF: Decentralizable Continuous Group Key Agreement with Fast Healing*. In 14th International Conference on Security and Cryptography for Networks - **SCN 2024**[AAN+22a]".

   [significant contributions to the protocol design and security analysis]

[†]Shared first authors
[‡]Authors in alphabetical order

The following list contains other works written during the PhD period but are not included in the thesis.

5. Benedikt Auerbach, Suvradip Chakraborty, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, Michael Walter, Michelle Yeo[‡]. *Inverse-Sybil Attacks in Automated Contact Tracing*. In Topics in Cryptology - CT-RSA 2021 - Cryptographers' Track, **CT-RSA 2021** [ACK+21].

6. Benedikt Auerbach, Charlotte Hoffmann, Guillermo Pascual-Perez[‡].*Generic-Group Lower Bounds via Reductions Between Geometric-Search Problems: With and Without Preprocessing*. In Theory of Cryptography. **TCC 2023** [AHPP23].

7. Benedikt Auerbach, Miguel Cueto Noval, Guillermo Pascual-Perez, Krzysztof Pietrzak[‡].*On the cost of post-compromise security in concurrent continuous group-key agreement*. In Theory of Cryptography **TCC 2023** [ACNPPP23].

8. Michael Anastos, Benedikt Auerbach, Mirza Ahad Baig, Miguel Cueto Noval, Matthew Kwan, Guillermo Pascual-Perez, Krzysztof Pietrzak[‡].*The Cost of Maintaining Keys in Dynamic Groups with Applications to Multicast Encryption and Group Messaging.* **TCC 2024** [AAB+24b].

9. Keitaro Hashimoto, Shuichi Katsumata, Guillermo Pascual-Perez[‡]. *Revisiting How to Authenticate Application Messages in MLS: More Efficient, Anonymous Blocklistable, and Post-Quantum*. To appear.

Table of Contents

List of Figures

List of Tables

CHAPTER 1

Introduction

> Architecture is politics,
> but it should not be understood
> as a substitute for politics
>
> Philip Agre, 2003

This thesis main motivation is *secure group messaging* (SGM) and, in particular, the understanding and improvement of the cryptographic key-exchange protocols that makes modern constructions practical and efficient.[1] Let us start introducing the topic through its different terms.

When we talk about **messaging**, we refer to the communication between individuals[2] through electronic devices, assuming the features that users have come to be expected when using modern applications like Whatsapp, Telegram, Slack, Signal or similar. Features such as immediate delivery of messages or the possibility of users communicating not being online at the same time. As one would expect, the term **group** refers to the fact that messaging does not only occur between two individuals, but within bigger sets of people (or devices, as one person often will own several devices within which conversations will be synchronized). While this extension is natural when it comes to the functionality of these apps, whose usability would be greatly reduced should only one-on-one conversations be supported, on a technical level it comes with a variety of

[1] This Chapter replicates, with permission, parts of our publications [KPPW+21, AAB+21b, AAN+22b, AAN+22a]

[2] Beyond individuals, communication between *smart* devices is also present in a growing number of situations, though we will not focus on that here.

challenges. In fact, maintaining the underlying cryptographic protocols efficient – and thus usable – beyond a relatively small number of users requires of new tools,[3] and research in developing these has been extensive in the last few years.

Finally, we wish these applications to be **secure**, meaning that we hope to have certain guarantees regarding the confidentiality, integrity and authenticity of the messages exchanged. One could see cryptography as (among other things) allowing us to bring the certainties of the real world into the online one. In the former, physical cues – seeing someone's face, hearing someone's voice, knowing we are in a secluded place without anyone else around, etc. – allow us to asses, for instance, who is speaking or who can learn: what is being spoken, when it was spoken, or even whether anything was spoken. Secure messaging protocols aim to provide us with similar guarantees regarding our online communications. These certainties about the context in which we are communicating give us agency over our participation in a conversation, empowering us to make an informed choice about what and how to communicate. A clear example of this is political talk, where users will use messaging apps as vehicle for communicating what they would not share in other, more public, social media [VV17]. Other examples include the need for a variety of collectives to protect their communications: journalists and their sources, activists in social movements, threatened minorities like refugees or people belonging to the LGBTQIA+ community, victims of targeted (e.g. domestic) violence, or employees at a company discussing a trade secret. The list could be endless. The existence of secure online communication tools is thus essential given our ever-growing reliance on online communications. As could be guessed, the intuitive translation between physical to online worlds will only go so far, since threat models in the latter can be quite different. Nonetheless, it gives us a good starting point.

We will dedicate the remainder of this Chapter to introduce the history and nature of SGM from a more technical lens (Section 1.1), discuss the social context in which it takes place (Section 1.2), present related work (Section 1.3), and outline the contributions of this thesis (Section 1.4).

1.1 Secure (group) messaging and CGKA

The protection of remote communication, which is vulnerable to interception in transit, is a longstanding goal. Early encryption tools designed to secure such communications can be traced back to Ancient Rome with the Caesar cipher, and possibly even earlier to

[3] While this is true already for powerful devices with fast network speeds, efficiency is key for demographics, particularly in the majority world [Ala08], that do not have such luxury. Data plans can be extremely expensive, with Zimbabwe being the most expensive in 2023 with 43.75 USD per GB [Cab23]. The median download speed also varies widely world-wide, with Cuba being the slowest country in May 2024, with a mere median 2.69 Mbps [Spe24].

Ancient Greece.[4] However, all the way from the Caesar cypher to the famous Enigma machine, used most notably by Germany during WWII, the use of these tools has been limited to a small number of people, mainly related to state or military affairs.

The advent of the internet, which rapidly became accessible to the average person, implied that most of the world population could engage in remote communication as we are used to today. E-mails became standard in businesses, universities and governments in the 1980s and 1990s. Though communication over the internet can much more easily be intercepted, secure alternatives like PGP for encrypting email were not popular, mainly for usability reasons [WT99].[5] And alternatives that came with the introduction of mobile devices, designed for communication with lower latency, like the Short Message Service (SMS), introduced in 1993, are not encrypted. Instant-messaging apps beyond SMS emerged in the late 2000s as a result of the popularization of smartphones, with Whatsapp being launched in 2009. Only a few years later, in 2013, chat apps surpassed SMS in global message volume for the first time [EM22]. This speaks of their success, as SMS boasted the staggering 6.1 trillion messages exchanged gobally in 2010 [107]. Nowadays, instant-messaging apps count billions of users worldwide, with Whatsapp alone having 2 billion. Early after the inception of Whatsapp, other apps, including TextSecure (now Signal), started deploying their own messaging protocols with adversarial threat models in mind. Around the same time, the Snowden revelations took place, bringing light to the extent ot the USA surveillance program, and acting as a catalyzer in the development of secure messaging protocols [EM22]. Currently, E2EE secure messaging protocols are ubiquitous tools in the daily life of billions of people. Indeed, E2EE is essentially the norm in instant messaging, with even companies like Meta implementing it into non-flagship products like Messenger and Instagram DMs [Met22].

A pioneering secure messaging protocol is Off-The-Record messaging (OTR) [BGB04], introduced in 2004. At the time, it not only provided end-to-end encryption, but also *deniable authentication* and *forward secrecy*, though at the cost of synchronicity, i.e. users needing to be online at the same time. Deniable authentication allows conversation participants to verify the authenticity of the messages while, at the same time, preventing any third parties from doing so. Forward secrecy, in turn, is a security feature that protects past messages from exposures of the key material, which could result from the adversary gaining physical access to the device, or hacking it. The general idea of providing security guarantees even in the presence of key exposures, which is particularly sensible given how long-lived sessions can be, would be key in the subsequent development of secure asynchronous messaging.

[4] Despite common folklore, the use of the scytale as an encryption tool in Ancient Greece is contested [Kel98], though it might have been used as an authentication method [Rus99].

[5] And are still not [RAZS16].

In fact, the main difference between modern secure messaging protocols and previous communication protocols is the adoption of regular key updates, designed to limit the effect of adversarial exposures of this key material. In particular, these key updates not only protect the confidentiality of already sent messages, as was the case of OTR, but, by infusing fresh randomness into the new keys, also allow for the confidentiality of future messages to be recovered. This is what is known as *post-compromise security* (PCS) [CCG16]. The most well-known and influential secure-messaging protocol achieving these notions, while allowing for asynchronous communication, is the Double Ratchet [MP16], introduced as part of the Signal protocol. The protocol is now employed by a great number of messaging apps like Signal, Whatsapp, Messenger or Skype. Other apps, most notably Telegram, have also developed their own secure-messaging protocols, and we will discuss this a bit more in detail in Section 1.2.

The double ratchet protocol is a two-party protocol, but a expected functionality of messaging apps is the support for group conversations. Almost all "1st generation" secure group messaging protocols made black-box use of this two-party protocol. However, this approach seems to unavoidably result in the complexity of (at least some critical) operations scaling linearly in the group size n.[6] This has resulted in practical limits in the groups sizes for deployed SGM protocols (often in 10s or low 100s and never more than 1000).[7]

Motivated by this, Cohn-Gordon et al. [CCG+18] initiated the study of SGM protocols whose complexity scales *logarithmically* in n. This work was the starting point for the Message Layer Security (MLS) working group at the Internet Engineering Task Force (IETF), back in 2018, with the objective to create a protocol scalable for groups of up to $50,000$ users. This process culminated in the publication of the first standard for group messaging in 2023 [BBR+23]. Even though some of the main standard development organizations bodies, like the IETF or W3C, have no legislative power and can only make recommendations, the reality is that they have an enormous impact in real-world deployments, with examples such as TLS[8] or TCP[9], the main protocols used to encrypt and transmit web communications, respectively, or, more recently, Privacy Pass[10] or OAuth [11], among many others. This impact is indeed wider than those of organizations like ISO or ITU, even though these do have closer affiliations with transnational organizations, like the United Nations [EM22, HCS20]. Thus, the

[6] In fact, this holds true even for the few 1st generation SGM protocols designed from the ground up with groups in mind [HLA19].

[7] https://9to5google.com/2023/01/18/google-messages-group-encryption-limit/ Accessed on 25.07.2024.

[8] https://datatracker.ietf.org/group/tls/about/

[9] https://datatracker.ietf.org/doc/html/rfc9293

[10] https://datatracker.ietf.org/wg/privacypass/about/

[11] https://datatracker.ietf.org/wg/oauth/about/

publication of this standard is bound to greatly shape the future of secure group messaging. And indeed it already has: MLS has either already been deployed, or soon will be, by companies and organizations like AWS, Cisco, Cloudflare, Google, Matrix, Meta, Mozilla, or Wire[12].

At the core of secure group messaging is a cryptographic primitive called *Continuous Group Key Agreement* (CGKA).[13] Intuitively, CGKA is to, say, SGM messaging what Key Agreement is to Public Key Encryption. That is, CGKA protocols capture many of the challenges involved in building practical higher-level E2EE secure applications (like messaging) while still providing enough functionality to make building such applications comparatively easy using known techniques [ACDT21]. Thus they present a very useful subject for research in the area.

1.1.1 Continuous Group Key Agreement

Continuous group key agreement (CGKA) was identified as the key primitive underlying group group messaging by Alwen *et al.* in [ACDT20, ACDT21]. In particular, the ART protocol [CCG+18], from which the MLS working group originated, can be seen as the first CGKA scheme, though it predates the formalization of the primitive. TreeKEM [BBR18] was proposed shortly after, substituting ART, and eventually becoming the CGKA employed by the MLS standard. Given the importance of the primitive, and of TreeKEM in particular, a variety of works have studied it, proposing alternative CGKA instantiations (often inspired by TreeKEM), analyzing the security of constructions, proving lower bounds, or targeting additional properties like CGKA for multiple groups, hiding metadata, etc. See Section 1.3 for references and a wider account of related work.

A CGKA protocol allows a set of users to maintain a shared key in an asynchronous setting, and protocol messages are relayed by an untrusted server. The operations CGKA must support are the users' addition and removal, and a key update functionality by which a user can rotate its secret key material so as to achieve forward secrecy and post-compromise security. More in detail, a CGKA has the following properties.

Dynamic membership. A CGKA protocol allows an evolving set of group members to continuously agree on a fresh symmetric key. Every time a new party joins, or an existing one leaves or refreshes (a.k.a. "updates") their cryptographic state, a new epoch begins in the session. Each epoch E is equipped with its own group key k_E, which can be derived by all parties that are members of the group during E. This contrasts with earlier primitives that model parties interactively agreeing on keys.

[12] `https://www.ietf.org/blog/mls-protocol-published/`

[13] Also referred to as *Group Ratcheting* [BDR20] or *Continuous Group Key Distribution* [BCK21].

PCFS. CGKA protocol sessions are expected to last for very long periods of time (e.g. years). Thus, as opposed to prior primitives like dynamic group key agreement, they must provide a property sometimes referred to as *post compromise forward security* (PCFS) [ACJM20]. This means that the group key of a target epoch should look random to an adversary despite having compromised any number of group members in both earlier and later epochs. This should hold true as long as the compromised parties either left the group or performed an update between their compromise and the target epoch. [14]

Asynchronicity. Most CGKA protocols today ([CCG+18, BBR18, ACDT20, KPPW+21, ACJM20, AJM22, AAN+22b, AFM24]) were designed with an asynchronous communication setting in mind (likely motivated by the application of secure asynchronous messaging). That is, parties may remain offline for extended periods of time, not ever actually being online at the same time as each other. Once they do come online, though, they should be able to immediately "catch up" and even initiate a new epoch (e.g. by unilaterally adding a new member to the group). This is in contrast to previous primitives modeling parties interactively agreeing on keys.

Untrusted network. In the spirit of distributed E2E security (and unlike, say, broadcast encryption, multicast encryption, or dynamic group key agreement) CGKA protocols must achieve all of the above without the help of trusted group managers or other specially designated trusted parties. Protocols are designed to communicate via an untrusted network which buffers protocol packets for parties until they come online again. Nevertheless, proposals to make multicast encryption amenable to the SGM setting by adding a key-updating functionality have been made [BDT22]. Additionally, several works [AHKM22, AAN+22b] assume a server than not only acts as a relay, but additionally provides extra functionalities, defining the *server-aided CGKA* primitive. Finally, several works [WKHB21a, AAN+22a] do not rely on a central server and assume a decentralized network.

[14] We note that PCFS is strictly stronger than providing the two more commonly discussed properties of Forward Security (FS) and Post Compromise Security (PCS). Indeed, a successful attack on an epoch E may require compromises *both* before *and* after E. Such an attack is neither an FS attack nor a PCS attack. Moreover, literature usually speaks informally of FS and PCS as separate notions asking that they both hold. Yet the notions do not necessarily compose and indeed protocols exist that have FS and PCS, but significantly worse PCFS [ACDT20]. Fortunately, all *formal* security definitions for CGKA we are aware of do in fact capture (some variation of) PCFS instead of treating FS and PCS separately.

1.1.2 Security Models: modeling the real world.

Designing systems secure against adversaries requires thinking in terms of worst-case scenarios, as there is no such thing as being secure against 95% of attacks. One cannot claim something secure as long as one attack that succeeds with non-negligible probability exists. At the same time, no model can abstract all real-world conditions [Box76] and capture all possible attacks. Further, the stronger the model, the more complex it becomes to prove any statement about a protocol in it. This posits a trade-off between the meaningfulness of a model and its practicality. The messaging literature contains a variety of models, varying across different axes such as the ability of the adversary to leak users secrets or randomness, to inject malicious messages, to control the delivery server (if one exists)... Further, the literature contains different definitions of what constitutes security, i.e., what properties should a protocol aim to achieve. This great amount of possible combinations gives raise to a plethora of security definitions, which are often incomparable.

In this section we aim to give an introduction to some of then main aspects present in different security models, together with an overview of those used in this thesis. We will start by reviewing the notions of adaptiveness and activeness. Then, we will shortly consider different frameworks used to prove security in the literature. We will end the section by discussing FS and PCS.

Adaptiveness. The literature distinguishes between *selective* and *adaptive* adversaries. In the selective case, an adversary is required to make all or some of its choices (here this means the sequence of operations and which key it is going to break) at the beginning of the security experiment, without seeing any public keys or the results of previous actions. While it is often more convenient to prove security in this setting, it is clearly unrealistic, since in the real world adversaries may adjust their behaviour based on what they observe during the attack. So obviously, security against adaptive adversaries is desirable. Most CGKA protocols have been proven adaptively secure, though some techniques for doing so are relatively recent, as we will discuss below.[15]

Activeness. One can classify adversaries with respect to their power to interact with the protocol during the attack. For example, the weakest form of adversary would be a

[15] Understanding adaptive security is still ongoing in some SGM-related areas though, like multi-party KEMs or mKEMs. This is a primitive allowing to reduce the communication cost of encrypting to multiple recipients, and which has been proposed as a building block of efficient SGM protocols by several works [KKPP20, HKP+21, AHKM22, AHK+23]. The most efficient and post-quantum-secure construction of mKEM is not known to be adaptively secure, though no attacks are known either [ACH+24]. While it is known how to turn it into an adaptively secure one, this transform doubles the public key and ciphertext size. Proving the former construction adaptively secure, or coming up with a better secure construction are interesting open problems.

passive adversary, i.e. an eavesdropper that only observes the communication but does not alter any messages. While the strongest notion would be an *active* adversary who can behave completely arbitrarily. Another notion is that of *insider* security, capturing safety against malicious group members. Earlier versions of MLS were susceptible to such a attack, where a user could create a malformed group and retain access to the group key even after being removed [AJM22].

Many works in the literature consider the intermediate notion of "partially" active adversaries[ACD19, CHK21, JMM19, KPPW+21, AAN+22b] (also somewhat implied by the model of [DV19], where communication must halt after an active attack). These adversaries can arbitrarily schedule the messages of the delivery server, and thus force different users into inconsistent states, but cannot inject messages, or force corrupted parties to arbitrarily deviate from the protocol and create malformed messages. The idea that an adversary can expose a user's key material, but not learn or be able to use the user's signature keys can be seen as quite a restrictive model. However, in practice, there are several mitigating aspects to this problem, that go beyond the difficulty of proving protocols actively secure. As was observed already during the design of MLS, in some real-world deployments of CGKAs fresh signature keys may be much harder to come by than simply locally generating new ephemeral key material. That is because each new signature key is typically bound to some external identity (like an account name) via some generic "authenticator" and this binding may be an expensive and slow process. E.g. a certificate that must be obtained manually from a CA. For this reason many CGKA protocols (including TreeKEM) explicitly permit *lite updates*; that is, updates which refresh all secret key material of the sender *except* for their signature keys. While lite updates are clearly not ideal from a security perspective, they do allow for frequently refreshing the remaining key material without being bogged down by the cost of certifying fresh signature keys. Moreover, TreeKEM and other CGKAs derive authenticity of packets not just from signatures but also by requiring senders to, effectively, prove knowledge of the previous epoch's group key. Thus, leaking a group members' signing keys does not automatically confer the ability to forge on their behalf. Indeed, if the victims all perform a lite update, a fresh epoch is initiated with a secure group key.

Security frameworks. When it comes to formalizing and proving security for CGKA (e.g. confidentiality, authenticity and group agreement), there are at least three approaches.

The first approach is *game-based*, where security is captured through a game in which the adversary gets to interact with the protocol. A proof essentially shows that it will be hard for the adversary to win the game, typically by reducing it to the security of the building blocks used in the protocol. This is the approach used to argue security of the protocols introduced in Chapters 3, 5 and 6, and indeed the approach used as

well by many other works, e.g. [CCG+18, ACDT20, ACDT21]. In this setting, there is a generic reduction from selective to adaptive adversaries that simply guesses what the adversary may choose (this is the approach effectively taken in [CCG+18]). However, this involves a loss in the advantage that is exponential (or even superexponential) in the size of the group. This means that in order to *provably* achieve meaningful security, one needs to set the underlying security parameter linear in the group size, which results in the update messages having size linear in the group size (since they usually consist of encryptions of secret keys). But the trivial construction based on pairwise channels also has message size that is linear in the number of group members, so such a security proof defeats the whole purpose of the design of efficienct CGKA protocols: having small update messages! Klein *et al.* [KPPW+21] are the first to prove meaningful security bounds against an *adaptive* adversary (and also introduced the basic GSD proof technique used in most subsequent game-based security proofs for adaptively secure CGKA). A somewhat different approach to the one above is taken by [ACDT21], which introduced the *history graph* technique to describe the semantics of any given CGKA executions. While it constitutes an extension of previous techniques that allows them to prove adaptive security of further schemes (like that of [ACDT20]), their game based notion still placed some restriction on when an active adversary could inject new packets. Game-based security comes with the advantage over other approaches of usually easier and more comprehensible proofs, achieving or even defining meaningful security against fully active adversaries is challenging, with no work in this setting doing so, to the extent of our knowledge.

A different approach to security notions is *simulation-based* definitions. This entails defining an ideal functionality, i.e., an idealized version of the protocol. Security is then argued by showing that an execution between the adversary and the real protocol is indistinguishable from one between the simulator and the ideal functionality. This was the route taken in [ACJM20] which presented the first ideal functionality for CGKA. Their notion captures security against powerful adaptive and fully active adversaries (i.e. they can deliver arbitrary packets without any restrictions) that can corrupt parties at will and even *set* their random coins.[16] Finally, building on that work, [AJM22] extend their adversaries to also account for how corrupt insiders might interact with a (very weak) PKI. The resulting notion is called *insider security*.

A third approach to defining adaptive and active security is through machine-checked *formal analysis*. This approach is first taken in [BBN19], who use an "event driven" language to define adaptive security for CGKA. E.g. authenticity is captured by roughly stating that if Alice believes a packet came from Bob then this must be preceded by an event where Bob sends such a packet. While currently unable to formally capture the

[16]The paper also proposes several protocols achieving this security notion at the cost of using impractical cryptographic primitives.

entirety of MLS, such an approach has proven very useful to identify subtle attacks that had otherwise been missed.

Forward Secrecy & Post-Compromise Security. FS and PCS are standard notions expected to hold in modern messaging protocols. In contrast to the two-party setting, formalizing these notions in the group setting is more nuanced. To start with FS, one natural approach is to require that a key is secure if all parties have performed an update before being corrupted. This is effectively what [CCG+18] does, and what we will adopt throughout this thesis. In contrast, [ACDT20] defines a stronger notion we refer to as *strong FS*. It requires keys to be secure as soon as possible, subject to not violating basic completeness of the CGKA protocol. However, this is only required in executions where protocol packets are delivered in the same order to all group members. Going even further, the work of Alwen *et al.* [ACJM20] introduces the notion of *optimal FS*, where, keys must become secure as soon as possible for *arbitrary* delivery order. To date, all protocols achieving these notions of stronger FS make use of either impractical primitives like hierarchical identity-based encryption (HIBE) [ACJM20], or of updatable public-key encryption (UPKE). Until the very recent work by Alwen *et al.* [AFM24], however, no UPKE constructions could satisfy the insider security notion introduced in [AJM22], which has become the goal in the literature. They achieve this through the enhanced notion of *joiner-secure UPKE*, which is secure even in the presence of forks in the key-updating execution.[17]

When it comes to PCS, the standard notion similarly requires that a key is secure if all parties performed a key-update after being corrupted. It is difficult to imagine a version of PCS requiring any less than this, since PCS requires new keys to be infused with new randomness, as the adversary is assumed to gain knowledge of the whole user's state after a corruption. Nevertheless, an alternative to this is introduced in the work of Fondevik, Hale and Tian, who propose the notion of *guardian PCS*, whereby a device might be paired with a "guardian" device that can update on its behalf. A different relaxation is the one introduced in [AAN+22b, AAN+22a] (Chapters 5 and 6), and generalized in [ANPPP23], where security is only required after each corrupted party updates a certain number of times from the time they were compromised (this number is fixed to $\log(n)$ in [AAN+22b] and to $\log(c)$ in [AAN+22a], where c is the number of corrupted parties).

[17] Their accompanying construction is not post-quantum secure, making it an interesting open problem to build a post-quantum, joiner-secure UPKE scheme. This would imply an SGM construction that is post-quantum and insider secure and has optimal FS.

1.2 The Social Context of Secure Group Messaging

Cryptographic work should not be separated from the social context in which it takes place, even if most of this work can be classified as theoretical. Many within the community have called for effectively acknowledging this and reassessing whether its work is aligned with its social implications, most notably Rogaway in his famous essay "The Moral Character of Cryptographic Work" [Rog15].[18] This is not new: a big influence in the development of cryptography are the cypherpunks, a strongly ideological group with motivations around individual privacy and autonomy, and institutional transparency [Hug93, And22]. Many tools and systems like Bitcoin, PGP, Tor and Wikileaks, which have had a profound impact on both society and the academic community, can be traced back to this community [Rog15]. This is also clear when considering E2EE, often discussed in the context of balancing the principles of "security through encryption" and "security despite encryption" [EM22]. E2EE is also the subject of significant regulatory scrutiny from governments [Kno23].[19]

On the one hand, cryptography is indispensable for a great number of systems in today's world, so cryptographers should ensure that the tools built serve their intended purpose. This starts with definitions, which play a very important role in what is visible, and how we approach working (in cryptography) on the problem [Rog19]. The choice of adversary, security goals, etc. are influenced by our view of what is desirable and realistic, and most often what is already part of our (academic cryptographic, in this case) community's assumptions and beliefs, meaning cryptography is strongly socially constructed [Rog09]. Even more, secure messaging tools and their building blocks (or any other cryptographic protocols) are "shaped by designers who make technical decisions that consider risk, threat models, [...], sociopolitical context and technical constraints. Decentralised versus centralised? Localisation versus globalisation? Anonymous or pseudonymous approaches? What counts as 'good' or 'desirable' security? The standardisation process itself, and design decisions about arrangements of architecture, are also arrangements of power" [EM22]. Indeed, when we develop or analyze protocols designed (or even amenable) for specific uses, such as for whistleblowers, we are making a normative statement about society – in this case, that whistleblowing is a valuable and beneficial activity.

On the other hand, it is important to be aware of the impact that cryptography (and more widely, technology) can realistically have. To begin with, internet access is very unevenly distributed globally with, e.g., 95% of the population of Italy having access to 5G while many territories around the world only have 2G networks (allowing just SMS and

[18] One initiative worth noting is the Re-Imagining Cryptography and Privacy Workshop (`https://recapworkshop.online/`).

[19] A comprehensive treatment of the social aspect and implications of encryption and secure messaging can be found in the book by Ermoshina and Musiani [EM22].

calls), or no network coverage exists at all [Cab23]. Further, many social and political issues are extremely complex and require of multi-disciplinary approaches that cannot rely solely on technology. The idea that technology alone will improve society or solve its problems can be very problematic [Rog24, GT24]. Additionally, our growing dependence on these tools also enable surveillance and political persecution [McL16, New19], as well as create new attack vectors on users [The19].

In this section we will discuss these issues applied to secure messaging more in depth. With this we also want to steer away from the techno-utopian self-indulgent view that cryptography will solve our social problems, while at the same time highlighting the many (and often subtly multi-layered) scenarios where it plays a big role. While the topic of this thesis is of a technical nature, we believe it a useful context. At the same time, we acknowledge our lack of expertise in this area and call for further multi-disciplinary approaches to the topic.[20]

1.2.1 Cryptography as a tool

We will start by identifying some settings in which encrypted messaging acts as a positive and important tool.[21]

Online communications have played a big role in social protest and activist movements over the last decade. Examples of this can be found in their use social media in the Gezi protests in Turkey in 2015 [HZ15], in the use of Whatsapp in Occupy Nigeria protests [URW18], in activist circles in Spain and Mexico [Tre20], or in the Hong Kong protests of 2014 and 2019, that were organized with the crucial help of social media and, in particular, through Telegram and internet forums [LC16, Tin20, KNC20, LCC20]. More generally, social media has played a major role in collective organization and action in the last decades [Cas12, MJHY15, Shi11, MNP18, LA10]. While some of the tools used by participants in the settings described above are not E2EE, there is a change in the last years towards relying on tools providing stronger security guarantees (though this comes with many hurdles, as we will discuss below). To take the example of Hong Kong, interviews with participants in the later protest movement showed that pseudonimity needs were very strong among protesters. This marked a shift in the security mindset with respect to earlier protests, particularly the Umbrella Movement from 2014, where an interviewee says "most was organized over Facebook" [ABJM21a]. Another example is the recent widespread use of E2EE apps like Whatsapp and the positive influence it has had on activism [GdZAACR21], and in users feeling more free to express their opinions, as showed by Valeriani and Vaccari [VV17], whose work "suggest[s] that MIMS [mobile instant messaging services] make a distinctive

[20] Such as this one: `https://social-foundations-of-cryptography.gitlab.io/2023/12/14/announcement-blog-post-martin/`.

[21] For the not so glamorous aspect of it see Section 1.2.3.

contribution to contemporary repertoires of political talk, with important implications for the quality and inclusiveness of interpersonal political discussion".

Encrypted communications are essential for journalists [LZR17, MCHR15, MFK16], who not only need to protect their data from adversarial entities but often rely on sources feeling safe to whistle-blow without personal repercussions [Di 20]. One of the most famous whistle-blowing tools is SecureDrop[22], created in 2013 to provide secure communication between journalists and sources. Recently, a new version incorporating end-to-end encryption and forward-secrecy guarantees for journalists has been announced to be in development [Sec24].

Additionally, many other collectives benefit from access to secure communication tools. A plethora of works have studied the privacy and information security needs for different groups, starting from civil society [SR16] and including transgender people [LHK+20], refugees [JCKT20, SLI+18], or undocumented migrants [GMS+18]. These works – and others, like that of Rosenbloom [Ros22] surveying Black Lives Matter (BLM) activists – suggest the populations studied have different needs that are often not satisfied and that need to be understood in order to design technologies meeting them [ABJM21a].

The question is thus: given the presence of such different threat models and needs, what to build and how to make sure it satisfies the intended purpose for which it is being built? And even more basic: is the cryptographic community focus on a few select tools like MLS or Signal the way forward? To what extent can a single tool satisfy the wishes of so many different groups?[23]

1.2.2 Building the right tools, and getting them used

The development of useful and secure digital tools is complex, as there is no such a thing as "uniform" user and tools are being used across different social contexts. Even within considered high-risk users, there is no such thing as a unique profile. On the one hand, there is a struggle between one's personal and "activist" identity [Bob07]. On the other, e.g., activists within the same group can be differentiated according to different roles and categories [Hor17]. Even the same groups might have different concurrent needs, such as protesters in Hong Kong, who differentiated between big open groups used for information distribution, which should be easily accessible, and smaller private groups for trusted core and collective decision making, where access is

[22] https://securedrop.org/

[23] When it comes to understanding what tools are needed by activists that might be targeted by bigger powers, an interesting approach is Ethan Zuckerman's famous "Cute Cat Theory of Digital Activism" [Zuc13].It posits that activists can benefit more from using mainstream platforms (those "primarily user to share cute cat pictures"), that bespoke ones that are easier to target.

heavily controlled[ABJM21a].[24] This paints a picture that is difficult to abstract in a cohesive threat model.

The point above is supported by the advice that digital security trainers give. Indeed, they take a person-based rather than tool-based approach, understanding that the context plays a huge role in what leads to a safer experience for the user. We can see this in the context of the recent war in Ukraine, where:

> *"The mere fact of having certain apps on one's phone (such as Signal, Tor or even Telegram) can raise suspicion and result in bodily harm or even life-threatening situations at the routine phone checks conducted by Russian soldiers. [...] digital security trainers aware of this context advice their high-risk users to use Whatsapp and Gmail instead of Signal or a PGP-encrypted form of email [...] The Ukrainian approach to security underlines that risk is relational and local. Security should be considered a multi-layered complex process in which the digital layer is just one of many."*
>
> (K. Ermoshina and F. Musiani [EM23])

The issue is further compounded by the commonplace discrepancies between the needs and wishes of users and the aims of designers and developers of SGM protocols and applications, as can be seen through the works of Ermoshina, Halpin and Musiani [EHM17, HEM18].[25] The first work [EHM17] relies on interviews with developers, privacy and security experts, and everyday users. The main lessons are: (a) protocol properties are often not understood by users[26] (b) high-risk users have different needs than low-risk ones, and (c) security trainers in countries with high-risk users due to, e.g., persecution of political dissidents, will suggest different practices and tools than those in low-risk countries. The second work [HEM18] follows on this and finds, for example, that "client device seizures are considered more dangerous than compromised servers by high-risk users", that "Key verification was important to high-risk users, but they often did not engage in cryptographic key verification, instead using other 'out of band' means for key verification" (this is further supported by [FK23]), or that developers placed too much priority on open standards, open-source or decentralization from the point of view of high-risk users.

Within the cryptographic academic community, this issue can be found as well in the attention that different protocols receive. For instance, a large amount of work

[24] Davidson, Virdia and Soezima recently proposed the notion of "semi-open group messaging" [DVS], addressing some of these needs.

[25] While the authors of both works are the same, we note that the second work presents the authors in a different ordering, with Halpin being first.

[26] Misconceptions and difficulties in understanding the security of applications are also shown by [ABJM21a] in the contexts of the 2019 Hong Kong protests; by [BSJCU21] in the context of BLM protesters; and, more widely, by many other studies [IRC15, ASSB+17, DNDS19]

has studied in the last years both the security guarantees of the double ratchet algorithm, and the problem of developing optimally secure protocols[BSJ+17, DV19, JS18, JMM19, PR18, ACD19]. This makes sense, since Signal's protocol boasts great security guarantees. Nevertheless, the reality is that Telegram is a major tool used in social movements, commonly perceived as the most trust-worthy app, and yet until recently it had not received much attention from the cryptography or information security communities [Kob18, ABJM21a].[27]

Another example is the prioritization of certain security properties by the academic community, with forward secrecy and post-compromise security being clear examples in the context of SGM. PCS is one of the main reasons for the development of a new generation of SGM protocols. It is motivated by the danger of an adversary gaining knowledge of the key-material of the device and thus retaining access to contents of the conversation from that point onwards. It is particularly concerned with such compromises happening without the user's knowledge, as updates are designed to be efficient enough to be able to occur with high regularity. However, it is unclear how common the latter type of compromises are and,[28] in turn, compromises that the user (and possibly other group members) becomes aware of do happen, and are a common cause of concern for e.g. activists [HEM18]. However, in the latter case users will often not rely on the security of key updates, but rather resort to more pro-active measures like re-creating the potentially compromised group.[29] On the other hand, forward secrecy in practice requires of the user using the disappearing messages feature to regularly delete messages from their phone. Indeed, a key-compromise not allowing to decrypt past ciphertexts is pointless if the underlying plaintexts are leaked together with the key. The disappearing messages feature, though present in many apps (e.g., Signal, Whatsapp, Telegram), is opt-in for most of them. The latter is an example of a great dilemma when building secure applications: the tension between usability and security.

Usability vs. Security As frustrating as it might be for the security community, the public will often favour usability over security. For example, adoption of messaging

[27]We should point out, nonetheless, that Telegram has been analyzed as of late by several works [MV23, AMPS22].

[28]Though this is to be expected since these compromises are, by definition, hard to identify, it shows the subjective nature of threat modelling.

[29]To make matters worse, many messaging apps that claim to have PCS were shown to be susceptible to cloning-attacks, where the adversary gains physical control of the device, clones its state, and attempts to impersonate the user [CFKN20].

platforms is often driven by features of the interface,[30], the ease of use,[31] the reputation of the app's creator, or the not-for-profit commercial status of the company, as is the case with Telegram [EM22].[32] And this makes total sense, oftentimes security is not really needed, whereas practicality is: the ability to make polls in a group might totally change the user experience. But sometimes security is relevant, and in these cases usability is often at odds with it. For example, it will be much harder to implement an anonymous SGM app if it needs to load an external service employed for sticker making. And average users will likely not make use of an app that requires to physically scan a QR code of your contact to verify their public key before communication can take place. Still, not caring about usability in these cases leads to "a 'forced responsibilisation' of users" [EM22] which implies the delegation of that responsibility to a few, hindering the development of collective approaches to security [AGRS15, Kaz15]. For example, usability and thus low adoption are still an issue in systems such as modern versions of PGP [RAZS16], despite being more than 30 years old and until now in development. Thus, developing secure applications needs both attention when designing and also when communicating to the public, making them aware of the choice they are making.

Ultimately, the reasonable conclusion is that it is unfeasible to design an app that can satisfy the needs of every user. A concrete example of this is the publication of the "Secure Messaging Scorecard" by the EFF [EFF16], aiming towards advising users on which apps to choose, and the criticism and revisions that followed [EFF18, EM22].

1.2.3 Cryptography and policy

In spite of the hurdles discussed in the development and deployment of SGM and related technologies, one could argue that existing technology is very advanced and that "[...] the most pressing issues of our time related to encryption may be not only legal and technical, but also social" [EM22]. Many national and international bodies have, in the last years, put forth legislative measures seeking to increase their oversight over security and privacy-related technologies.[33] An example of regulation of E2EE is

[30] Iranian users explained in an interview-based study that the possibility to use "stickers representing Muslims in their everyday environment [...] was a very important feature that differentiates Telegram from 'first world' apps that only focus on Western lifestyles and emoticons" [EM22].

[31] Users prefer convenience and, though are generally interested in using secure tools, are often unclear as to their personal benefit, or would not want to use them regularly [RAH+16].

[32] The reputation of Telegram founders, particularly Pavel Durov, is partly due to the russian government declaring him *persona non-grata* after refusing to collaborate with Russia's Federal Security Service. This ideological factor when choosing Telegram can be seen by its massive increase in downloads when Facebook bought Whatsapp [EM22]. This is in contrast to the fact that group conversations in Telegram are not E2EE, and nor are one-on-one conversations by default.

[33] A great and somewhat related example of governmental regulation targeting technology is the General Protection Data Regulation (GDPR), issued by the European Union in 2016. The current focus in these areas follows unequivocal statements like that of Meglena Kuneva, European Consumer

the Yarovaya law [Luh16], going into effect in 2018 in Russia, which requires telecom providers to store the contents of voice calls, data, images and text messages and asks any company using encrypted data to allow the Federal Security Service of the Russian Federation (FSB) access to its content.[34] Another example is the call in 2018, by state members of the 'Five-Eyes' (Australia, Canada, New Zealand, United Kingdom and United States), together with India and Japan, for technology companies to make it possible for law enforcement to have access to content, where justified by law [OoPA]. The will to regulate E2EE is reasonable given the dark side of messaging applications, which can be used to harass others [TAB+21], spread violence and encourage mob lynchings [Muk20, Aru19], to spread misinformation [Sta21, Bel], or as a vehicle to share other harmful content such as Child Sexual Abuse Material (CSAM).[35]

An issue when discussing such regulation is that it is hard to agree on regulation when not agreeing on definitions. For example, defining E2EE properly is not a trivial task, and encompasses subtleties beyond the intuitive combination of authenticity and confidentiality. Indeed, the formalization of these notions (e.g. through primitives like Authenticated Encryption (AE)), is not enough to accurately abstract real-world needs, where one additionally needs to properly define what we understand by an *end* [HK22]. This is shown by companies claiming to offer E2EE services while actually providing just client-to-server encryption (the case of Zoom [MSR20] is a notable one). Another revealing example of this issue is the controversy regarding content moderation in encrypted communications, brought about to the public discourse by proposals like Apple's client-side scanning (CSS) solution to the distribution of CSAM [App21]. Roughly speaking (see, e.g., [AAB+24a] for a technical discussion), this would allow external parties to learn whether the contents of a user's encrypted data matches those of a centralized database, which is a substantial modification from the folklore understanding of E2EE, where no one except the users communicating can learn anything about the contents of the transmitted data. Yet, if the contents of said database are trusted and limited, it could be argued that it would be reasonable to refer to it as E2EE. What is hence the right definition for E2EE that can balance "security through encryption and security despite encryption"? This is an ongoing discussion across different parts in the globe, as evidenced by the recent postponement of a proposal along these lines by the EU [Tim]. The preference for CSS by many stakeholders is often borne out of the perceived lack of alternatives [BGD+24]. Nevertheless, whether technical solutions are the best approach to solving such complex social challenges is another entirely different and crucial question [FN21, Kno24, Tro23, EE].

Commissioner, from 2009: "Personal data is the new oil of the internet and new currency of the digital world" [Kun09].

[34] This triggered Telegram to famously leave the country [EM21].

[35] Mitigating some of these downsides through technical means is the object of ongoing research [IAV22].

Interoperability One last notable piece of legislation is the Digital Markets Act (DMA)[36] law by the EU, that mandates, among many other things, for messaging apps with enough share of the market to interoperate [RS23]. This is aimed at preventing monopolies from the so-called *gate-keepers*.[37] Furthermore, this would potentially help prevent the *silo effect* [SH15] of having the user base fragmented across different apps, which has been one of the main drawbacks towards the adoption of E2EE apps [EM22, ASKP+17]. However, doubts have been raised about the potential contradiction between interoperability and the preservation of security [BA23]. While some solutions have been suggested [LGGR23] our collective understanding of this problem is limited.[38] Further, a potential solution such as the adoption of a common standard like MLS by most messaging apps seems unlikely. This unification would also come at the cost of reducing the user's ability to choose based on their threat model.

1.3 Related Work

In Section 1.1 we discussed some works and how they fit in understanding the different security models and history of the development of CGKAs. In this section we additionally mention some works that are relevant to understand the group messaging landscape. Further ones that are relevant mostly for a particular section of this thesis will be mentioned in the corresponding chapter.

Even though CGKA is relatively recent, a closely related and much older primitive is that of Group Key Exchange (GKE), which allows a fixed group of users to derive a common key. These can be traced to early publications like [ITW82, BD95]. In contrast to CGKA, GKE protocols do not target PCS and are designed for the synchronous setting. That is, they are highly interactive e.g. requiring all parties to contribute to any one operation via interactive rounds. Initial GKE results were followed by a long list of works exploring additional features; notably, supporting changes to group membership mid-session (aka. Dynamic GKE) [BCP02, DB05]. A paper by Poettering *et al.* [PRSS21] surveys the different models used in the GKE literature, including some CGKA ones.

Another related notion is that of Multicast Encryption (ME) [Pan07], which allows a changing group of users to maintain a common key with the help of a trusted group manager. Logical Key Hierarchies (LKH) [WHA98, WGL98, CGI+99] were introduced as efficiency ingredients for building ME protocols. Ratchet trees, the graph structures

[36] https://digital-markets-act.ec.europa.eu/index_en

[37] Whatsapp and Messenger were identified as the only gatekeepers within the instant messaging categoryhttps://digital-markets-act.ec.europa.eu/gatekeepers_en.

[38] This problem is the focus of the IETF "More Instant Messaging Interoperability" working group (https://datatracker.ietf.org/wg/mimi/about/.)

used by TreeKEM and most existing CGKA protocols, can be seen as simple adaptations of LKH to the CGKA setting.

The study of CGKA based protocols (group key agreement protocols with the explicit goal of PCS, asynchronous communication and no trusted parties) was initiated by the ART protocol [CCG+18], based on which the first version of MLS [BBR+23] was built shortly before transitioning to TreeKEM [BBR18].

Many variants of TreeKEM, aiming to improve it across different axes have been proposed since its inception, starting with Tainted TreeKEM [KPPW+21], presented in Chapter 3, and which presents an alternative method to remove or add users to the group. The recent work of Chevalier *et al.* [CLMP24] analyzes the efficiency deficits of using binary left-balanced trees and proposes optimized algorithms for adding users and expanding ratchet trees. Metadata protection is an important aspect of privacy. Signal employs techniques such as Sealed Sender [Sig18] and Private Groups [CPZ20], which hide the identity of senders and group members from the server, respectively. Metadata-hiding modifications of MLS have been proposed in [HKP22], and [BRT23] studies anonymous messaging using mesh networks. In response to the interoperability mandates by the European Union's DMA, [LGGR23] identifies its effects on encrypted messaging systems, putting forth a proposal for a protocol, as well as open problems. The paper by Bienstock, Dodis and Tang [BDT22] studied the enhancement of ME schemes for their application to group messaging. Recently, Alwen, Fuchsbauer and Mularczyk [AFM24] introduce a stronger notion of updatable PKE that is more appropriate for its use in multi-party settings and, in particular, secure group messaging.

In [ACJM20] zero-knowledge proofs are used to improve the robustness of CGKA protocols. The approach was made a bit more practical in [DDF21] by, amongst other things, introducing tailor-made ZK proofs. Very recently, [EKN+22] initiated the study of membership privacy for CGKAs. Other works have focused on tools for cryptographic administration of group membership [BCV23], or resilience against network splits [AMT23]

The work of [KKPP20] initiated the study of post-quantum primitives for CGKA by building primitives designed for use in TreeKEM (and similar CGKAs). This was followed up by the work of Hashimoto *et al.* [HKP+21], who propose the use of multi-recipient PKE in order to improve the download cost of users, at the cost of linear size commits. This primitive, in combination with reducible signatures, is also used by Alwen *et al.* in [AHKM22]. This work introduces the notion of server-aided CGKA as well as notable efficiency improvements by greatly reducing constant factors (though communication stays similar to TreeKEM's asymptotically). Further improvements over previous results are given in [AHK+23]. The notion of multi-recipient KEM has recently been proposed for standardization at NIST [ACH+24].

The later versions of TreeKEM support a certain degree on concurrency between group operations. Outside of TreeKEM, concurrency was initially approached by Weidner [Mat19], who proposed the Causal TreeKEM protocol. Later, it was more formally analyzed by Bienstock *et al.* [BDR20], who study the trade-off between PCS, concurrency and communication complexity, showing a lower bound for the latter and proposing a close to optimal protocol efficiency-wise, though in a static-group model, and with weaker security. A further paper by Weidner *et al.* [WKHB21b] proposes a decentralized and concurrent protocol, at the cost of linear communication cost for updates. Recently, Cong *et al.* [CEST24] proposed the so-called *key-lattice* framework and a concrete instantiation, allowing for commutative updates, which also incurs linear communication costs. Processing concurrent operations, together with a relaxed notion of PCS, is also the aim of protocols CoCoA [AAN+22b] and DeCAF [AAN+22a], presented in Chapters 5 and 6, respectively.

A variety of papers have been devoted to understanding TreeKEM and the wider CGKA landscape. First, we have papers showing lower bounds on the communication cost that CGKA protocols satisfying certain properties need to achieve. In this setting, the work of Alwen *et al.* [AAB+21b] show lower bounds on the communication cost of updates in potentially overlapping groups, along the way proving that the logarithmic cost of a single update in ratchet trees is optimal. Bienstock, Dodis and Rösler [BDR20] give a lower bound on the cost of a group healing from a compromise through concurrent updates in two rounds of communication (which is the minimal number of round needed without the need of some powerful primitive like multi-party non-interactive key-exchange). This lowerbound is generalized by Auerbach *et al.* [ANPPP23] to the setting where PCS can be achieved in $k \geq 2$ rounds of communication. Further, the recent paper by Anastos *et al.* [AAB+24b] presents a lower bound on the cost of group operations that change the set of group members, showing that current solutions are optimal. A paper by Bienstock *et al.* [BDG+22] proves no CGKA can be instantiated in a black-box way from PKE with sublinear cost, and that no optimal (with respect to any sequence of operations) protocol exists.

Further, we have papers that have attempted to formalize TreeKEM in terms of smaller, simpler to analyze, building blocks. From this lens, Alwen *et al.* [ACDT21] formalize secure group messaging and cast it modularly in terms of the primitives CGKA, forward-secure group AEAD and PRF-PRNG. A somewhat different abstraction of MLS is put forth by Wallez *et al.* [WPBB23], which splits it into TreeKEM, TreeDEM, similar to the concept of FS-GAEAD above, and TreeSync, a new abstraction capturing the guarantees regarding state consistency between users. Finally, others have studied the security guarantees of TreeKEM, along the way extending previously existing security models. In terms of security, the first security proof for any CGKA was for ART in [CCG+18]. Their proof has an exponential loss against *adaptive* adversaries. The first proof of adaptive security with sub-exponential loss (in fact, polynomial in the

random oracle model) for a variant of TreeKEM was given in [KPPW+21]. While their security proof captured adversaries who can make adaptive choices, it did not capture fully active adversaries who can arbitrary deviate from the protocol specification and send malformed messages. Subsequent works [ACJM20] and [AJM22] propose stronger security models, allowing the adversary to set the random coins of parties, and to corrupt and impersonate them, respectively. Formal analyses of TreeKEM's security were carried out in [BCK21], [BBN19] and [WPBB23]. In the multi-group setting, Cremers *et al.* [CHK21] study the PCS guarantees earlier versions of TreeKEM provided for users belonging to different groups. The work of Alwen *et al.* [ACDT20] analyzes the forward security guarantees of TreeKEM, proposing an improvement based on updatable PKE, which achieves optimal FS: parties need only process an update from any other party, as opposed to issuing one update. This work was the first one to introduce the definition of CGKA.

Finally, a number of recent works have focused on assessing the security of existing messaging apps, like Bridgefy [ABJM21b, AEP22], Telegram [AMPS22], DeltaChat [SMP24], Threema [PST23], or the Sender Keys protocol, used by both Signal and Whatsapp [BCG23].

.

1.4 Outline and Contributions

The rest of this thesis is structured as follows. Chapter 2 covers some preliminary concepts, including the formal definition of CGKA and its security, as well as an overview of ART and TreeKEM. Chapter 3 introduces the Tainted TreeKEM protocol, an alternative for executing membership changes in the group. Chapter 4 studies the setting of multiple (possibly overlapping groups) and the protocol-design space in it. Chapter 5 introduces the CoCoA protocol, designed to enable concurrent group operations without efficiency downsides. Chapter 6 introduces the DeCAF protocol. As CoCoA, it allows for concurrency, but introduces different trade-offs with respect to it. More in detail, our contributions are the following.

Chapter 3: Alternative handling of Dynamic Operations. The main method for executing membership changes in TreeKEM and most CGKA protocols is through a technique called *blanking* (see Section 2.5), which effectively erases from the common state of group users that would otherwise be used to efficiently communicate state changes to the group. The obvious downside of this approach is that it increases the communication cost of future group operations. In Chapter 3 we present Tainted TreeKEM (TTKEM), a variant of TreeKEM that employs a different approach, which we termed *tainting*, to handle these operations. Tainting does not delete keys and

instead replaces them by new ones. While doing so preserves the structure of the common group state, in order to preserve security certain operations will still incur an overhead in communication. This presents a trade-off between TreeKEM (and similar protocols) and TTKEM. We complement the description of tainting and the resulting protocol with a comparison based on simulations, showing that our protocol can be more efficient in several natural scenarios.

Chapter 4: Efficiency Optimizations through Overlapping Groups. Users of a messaging service will typically be part of different groups, often with overlapping membership sets. In Chapter 4 we initiate the study of protocols that make use of these overlaps to improve on the overall efficiency of the system.

Our contributions in this regard are twofold. On the one hand, we look are the problem from a more theoretical lens in order to gauge the potential that such a solution can have over the trivial one where the overlap is not exploited, in the setting where the number of users grows to infinite. Here, we prove a lower bound on the cost of key updates and complement it with an algorithm that matches this bound. On the other hand, we approach the problem from a more practical side, taking into account concrete efficiency, and we propose an algorithm that is better suited for certain concrete set systems, analyzing its efficiency. We complete the chapter by showing a lower bound on the average update cost of arbitrary CGKA and ME schemes in a symbolic model. This lower bound both improves and extends that of Micciancio and Panjwani [MP04] for ME. In particular, it extends it to both the settings of CGKA and multiple groups, and lifts it from worst to average communication complexity.

Chapters 5 and 6: Concurrent Key Updates and their Effect on PCS. For groups containing many users, key updates occur with a high frequency, which creates a need to handle the case where several parties concurrently issue such a protocol message. Schemes allowing for several members to rotate their keys in a single communication round not only solve this problem, but further open the door to stronger PCS guarantees, as security can be restored more quickly following a compromise. This was already identified as an important aim in the original version of TreeKEM, and was one of the factors leading to the introduction of the Propose & Commit paradigm in later versions. While this approach and others did (partially) solve the concurrency issue, they did so at a high cost in communication complexity. In Chapters 5 and 6 we propose two schemes that allow for members to rotate their keys concurrently and efficiently and without degrading future communication complexity. In doing so, we introduce a new, more flexible notion of PCS that only requires security to be restored after a certain number of updates per party have been performed.

In Chapter 5 we introduce CoCoA, a protocol which supports concurrent operations

without the need of the P&C paradigm and thus without the need of blanking. We show that users in CoCoA achieve PCS in a number of rounds that is logarithmic in the number of parties updating their keys. Before CoCoA no scheme was known that could heal from a compromise without incurring either a linear number of communication rounds (non-concurrent schemes) or a degradation in subsequent communication (concurrent ones).

Further, in Chapter 6 we introduce DeCAF, a variant of CoCoA employing a form of updatable public key encryption, and which achieves PCS in a number of rounds that is independent of the number of concurrent updating members, and in fact is only logarithmic in the number of state compromises (which will typically be much lower than the number of users).

Chapter 5: Reduced Download Complexity. The bulk of communication cost in secure group messaging is concentrated around downloading, since users will typically witness (and download) operations from other group members linearly many (in the group size) more times than issue them. An additional contribution of Chapter 5 is the introduction of the notion of *partial states*, a building block in CoCoA, which allows users to store (and download) only a fraction of the information uploaded by other users. This effectively reduces the download cost per-user per-round from linear to logarithmic in the size of the group. CGKA schemes preceding CoCoA required either a linear upload communication cost, or a linear total download communication cost (across the process of healing from a compromise). CoCoA was the first scheme introduced that achieved poly-logarithmic communication cost on both sides.

CHAPTER 2

Preliminaries

2.1 Notation

- Throughout the thesis $\log$ denotes the logarithm with respect to base 2.
- Given natural numbers $a, b \in \mathbb{N}$ and $a \leq b$, we write $[a, b]$ to denote the set $\{a, a+1, \ldots, b-1, b\}$. Further, we use the notation $[a]$ as shorthand for $[1, a]$, and $[a]_0$ for $[0, a]$.
- We write $a \| b \in \{0,1\}^*$ to denote the bit-string resulting from concatenating a and b when viewed as bit-strings.
- For a probability distribution X, we write $x \leftarrow X$ to denote that x was randomly sampled from X. Likewise, for a set X, $x \leftarrow X$ denotes sampling uniformly at random from X. Further, for a (possibly randomized) algorithm X, $x \leftarrow$ X denotes the output of X. Finally, we also express that variable x gets assigned value v by $x \leftarrow v$. Given a set S, we write $S \xleftarrow{\cup} s$ as shorthand for $S \leftarrow S \cup \{s\}$.
- We write $X \approx_c Y$ to express that X and Y are to computationally-indistinguishable probability distributions.
- Finally, we say that a function negl : $\mathbb{N} \to \mathbb{R}_{>0}$ is negligible if for every polynomial p there is an integer N_p such that, for all $n > N_p$, $|\mathsf{negl}(n)| < 1/p(n)$.

2.2 Cryptographic building blocks

In this section we define some basic cryptographic functions we will use throughout the thesis. For more detailed explanations we refer the reader to [KL14], from which most of these definitions were adapted.

2.2.1 MACs

Defintion 2.2.1. *A* message authentication code *(MAC) consists of three probabilistic polynomial algorithms (*$\mathrm{MAC.Gen}, \mathrm{MAC.Tag}, \mathrm{MAC.Ver}$*) such that:*

1. *Algorithm* $\mathrm{MAC.Gen}$ *takes as input the security parameter* 1^n *and outputs a key* k *with* $|k| \geq n$.
2. *Algorithm* $\mathrm{MAC.Tag}$ *takes as input a key* k *and a message* $m \in \{0,1\}$*, and outputs a tag* t.
3. *The deterministic algorithm* $\mathrm{MAC.Ver}$ *takes as input a key* k*, a message* m *and a tag* t *and outputs a bit* b.

We require that for every n*, every key* k *output by* $\mathrm{MAC.Gen}(1^n)$*, and every* $m \in \{0,1\}^*$*, it holds that* $\mathrm{MAC.Ver}_k(m,t) = 1$.

To define security, consider the following experiment Mac-forge$_{\mathrm{Adv,MAC}}(n)$.

1. *A key* k *is generated by running* $\mathrm{MAC.Gen}(1^n)$.
2. *The adversary is then given input* 1^n *and oracle access to* $\mathrm{MAC.Tag}_k(\cdot)$*. The adversary eventually outputs* (m,t)*. Let* $\mathcal{Q}$ *denote the set of all queries that* $\mathcal{A}$ *submitted to its oracle.*
3. $\mathcal{A}$ *succeeds if and only if* $\mathrm{MAC.Ver}_k(m,t) = 1$ *and* $m \notin \mathcal{Q}$*. In that case the output of the experiment is defined to be* 1.

We say that a MAC $\mathrm{MAC} = (\mathrm{MAC.Gen}, \mathrm{MAC.Tag}, \mathrm{MAC.Ver})$ *is secure if, for all probabilistic polynomial adversaries* $\mathcal{A}$*, there is a negligible function* negl$(\cdot)$ *such that*

$$\Pr[\text{Mac-forge}_{\mathrm{Adv,MAC}}(n) = 1] \leq \text{negl}(n)$$

.

2.2.2 PRFs

We will use the following definition for pseudorandom functions. It is an easy exercise to prove that the below definition is equivalent to the standard textbook definition of PRFs (i.e., only a polynomial loss in security is involved by the respective reductions). We show this, since it is the definition that the security proof of Tainted TreeKEM (see Chapter 3) needs, even if proof in question is not included in this thesis.

Defintion 2.2.2. *[Pseudorandom function, alternative definition] Let* $H : \{0,1\}^n \times \{0,1\}^n \to \{0,1\}^n$ *be a keyed function. We define the following game* $\mathsf{PRF}(n)$*:*

1. *A key* $k \leftarrow \{0,1\}^n$ *is chosen uniformly at random and the adversary is given access to an oracle* $H(k,\cdot)$*.*
2. *The adversary outputs a string* $x \leftarrow \{0,1\}^n$*, a uniformly random bit* $b \leftarrow \{0,1\}$ *is chosen and the adversary receives either* $H(k,x)$ *in the case* $b = 0$*, or* $y \in \{0,1\}^n$ *uniformly at random if* $b = 1$*.*
3. *The adversary outputs a bit* b'*. If* x *was never queried to the oracle* $H(k,\cdot)$ *and* $b' = b$*, then the output of the game is* 1*, otherwise* 0*.*

We call H (ε, t)-pseudorandom *if for all adversaries* A *running in time* t *we have*

$$\mathsf{Adv}_{\mathsf{PRF}}(\mathsf{A}) := |\mathsf{Pr}[1 \leftarrow \mathsf{PRF}(n)|b = 0] - \mathsf{Pr}[1 \leftarrow \mathsf{PRF}(n)|b = 1]| < \varepsilon.$$

2.2.3 PKE

In order to encrypt data, MLS makes use of authenticated encryption with associated data (AEAD) [ACDT21]. Nevertheless, the more straightforward definition of standard public-key encryption (PKE) will be sufficient for this thesis, as we make a number of simplifications. Since the security model under which the different protocols are proven secure does not consider fully active adversaries (see Section 2.4), we will only require IND-CPA security from the PKE scheme.

Defintion 2.2.3. *A* public-key encryption scheme *is a triple of probabilistic algorithms* $(\mathrm{PKE.Gen}, \mathrm{PKE.Enc}, \mathrm{PKE.Dec})$ *such that:*

1. *Algorithm* $\mathrm{PKE.Gen}$ *takes as input the security parameter* 1^n *and outputs a pair of keys* $(\mathrm{pk}, \mathrm{sk})$*, where* pk *defines some message space* $\mathcal{M}_{\mathrm{pk}}$*.*
2. *Algorithm* $\mathrm{PKE.Enc}$ *takes as input a public key* pk *and a message* $m \in \mathcal{M}_{\mathrm{pk}}$*, and outputs a ciphertext* c*.*

3. *Algorithm* $\mathrm{PKE.Dec}$ *takes as input a private key* sk *and a ciphertext* c *and outputs a message* m *or a special symbol* $\perp$*, noting failure.*

We require that, except with negligible probability over the randomness of $\mathrm{PKE.Gen}$ *and* $\mathrm{PKE.Enc}$*, we have* $\mathrm{PKE.Dec}_{\mathrm{sk}}(\mathrm{PKE.Enc}_{\mathrm{pk}}(m) = m$ *for any* $m \in \mathcal{M}_{\mathrm{pk}}$*.*

Defintion 2.2.4. *Consider the following experiment* $\mathsf{PubK}^{\textit{IND-CPA}}_{\mathcal{A},\Pi}(n)$ *defined with respect to an adversary* $\mathcal{A}$ *and a PKE scheme* $\mathrm{PKE} = (\mathrm{PKE.Gen}, \mathrm{PKE.Enc}, \mathrm{PKE.Dec})$*.*

1. $\mathrm{PKE.Gen}(1^n)$ *is run to obtain keys* $(\mathrm{pk}, \mathrm{sk})$*.*
2. $\mathcal{A}$ *is given* pk*, and outputs a pair of equal-length messages* $m_0, m_1 \in \mathcal{M}_{\mathrm{pk}}$*.*
3. *A uniform bit* $b \leftarrow \{0, 1\}$ *is chosen, and then a ciphertext* $c \leftarrow \mathrm{PKE.Enc}_{\mathrm{pk}}(m_b)$ *is computed and given to* $\mathcal{A}$*. We call* c *the* challenge ciphertext*.*
4. $\mathcal{A}$ *outputs bit* b'*. The output of the experiment is* 1 *if* $b' = b$*, and* 0 *otherwise.*

We say PKE *is* IND-CCA secure *if, for all adversaries* $\mathcal{A}$*, there is a negligible function* negl *such that:*

$$\Pr[\mathsf{PubK}^{\textit{IND-CPA}}_{\mathcal{A},\mathrm{PKE}}(n) = 1] \leq \frac{1}{2} + \mathsf{negl}(n)$$

2.2.4 Signature Schemes

Defintion 2.2.5. *A* signature scheme SIG *consists of three probabilistic polynomial-time algorithms* $(\mathrm{SIG.Gen}, \mathrm{SIG.Sig}, \mathrm{SIG.Ver})$ *such that:*

1. *Algorithm* $\mathrm{SIG.Gen}$ *takes as input the security parameter* 1^n *and outputs a pair of keys* $(\mathrm{svk}, \mathrm{ssk})$*, which we will refer to as the* verification *and* signing *keys, respectively.*
2. *Algorithm* $\mathrm{SIG.Sig}$ *takes as input a singing key* ssk *and a message* m *from some message space (possibly dependent on* svk*) and outputs a signature* $\sigma \leftarrow \mathrm{SIG.Sig}_{\mathrm{ssk}}(m)$*.*
3. *Deterministic algorithm* $\mathrm{SIG.Ver}$ *takes as input a verification key* svk*, a message* m *and a signature* σ*. It outputs a bit* $b = \mathrm{SIG.Ver}_{\mathrm{svk}}(m, \sigma)$*, with* $b = 1$ *meaning the signature is valid,* $b = 0$ *meaning invalid.*

It is required that, except with negligible probability over the output of $\mathrm{SIG.Gen}$*, it holds that* $\mathrm{SIG.Ver}_{\mathrm{svk}}(m, \mathrm{SIG.Sig}_{\mathrm{ssk}}(m)) = 1$ *for every* m*.*

Defintion 2.2.6. *Consider the following experiment* Sig-forge$_{\mathcal{A},\mathrm{SIG}}(n)$ *given a signature scheme* $\mathrm{SIG} = (\mathrm{SIG.Gen}, \mathrm{SIG.Sig}, \mathrm{SIG.Ver})$ *and adversary* $\mathcal{A}$*:*

1. $\mathrm{SIG.Gen}(1^n)$ *is run outputting* $(\mathrm{svk}, \mathrm{ssk})$.
2. *Adversary* $\mathcal{A}$ *is given* svk *and oracle access to* $\mathrm{SIG.Sig}_{\mathrm{ssk}}(\cdot)$*. The adversary then outputs* (m, σ)*. Let* $\mathcal{Q}$ *denote the set of all queries that* $\mathcal{A}$ *made to its oracle.*
3. $\mathcal{A}$ succeeds *if and only if* $\mathrm{SIG.Ver}_{\mathrm{svk}}(m, \sigma) = 1$ *and* $m \notin \mathcal{Q}$*. In this case, the output of the experiment is defined to be* 1.

We say that SIG *is existencially-unforgeable under an adaptive chosen-message attack, or EU-CMA secure, if, for all probabilistic polynomial-time adversaries* $\mathcal{A}$*, there is a negligible function* negl *such that:*

$$\Pr[\text{Sig-forge}_{\mathcal{A},\mathrm{SIG}}(n) = 1] \leq \mathsf{negl}(n)$$

2.2.5 The Random Oracle Model

In order to simplify and make more approachable (some of) the security proofs in this thesis, we will employ the *random oracle model (ROM)*, a commonly used idealized model which treats hash functions as truly random functions. A hash function is thus a map $\mathsf{H} : \{0,1\}^* \mapsto \{0,1\}^l$ taking arbitrary strings to ones of fixed length that the adversary will only have oracle-access to. That is, given an input $x \in \{0,1\}^*$, the only way to obtain output y such that $y = \mathsf{H}(x)$ is by querying the oracle on x.

The claim is not that a real random oracle exists, and indeed it is known that there exist schemes proven secure under the ROM that are insecure under any instantiation of hash function in the *standard model* (where no random oracle is present) [MRH04]. Rather, the hope is that the employed hash function is "sufficiently good" at emulating a random oracle, so that security of the real-world instantiation of the scheme follows from the (idealized) security proof [KL14].

2.3 Continuous Group-key Agreement

To begin with, we define the notion of continuous group-key agreement (CGKA). Parties participating in the execution of a CGKA protocol will maintain a local state γ, allowing them to keep track of a common group information and, in particular, to derive a shared secret. Parties will be able to add and remove users to the execution, and to rotate the keys along sections of the tree, thus achieving FS and PCS. Our definition is essentially that of [AAN+22b], which itself can be seen as a generalization of the

definition of [KPPW+21], with the main difference that operations can be concurrently processed by users through a single message curated by the server. The (potentially stateful) server works in rounds, collects operations into batches and sends (potentially processed versions of) them out at the end of each round (note that setting the batch size equal to 1 would just return the definition from [KPPW+21]).

Defintion 2.3.1 (Asynchronous Continuous Group-key Agreement)**.** *An* asynchronous continuous group-key agreement *(CGKA) scheme is an* 8*-tuple of algorithms* $\mathrm{CGKA} = (\mathrm{CGKA.Keygen}, \mathrm{CGKA.Init}, \mathrm{CGKA.Add}, \mathrm{CGKA.Rem}, \mathrm{CGKA.Upd}, \mathrm{CGKA.Dlv}, \mathrm{CGKA.Proc}, \mathrm{CGKA.Key})$ *with the following syntax and semantics:*

Key Generation**:** *Fresh InitKey pairs* $((\mathrm{pk}, \mathrm{sk}), (\mathrm{ssk}, \mathrm{svk})) \leftarrow \mathrm{CGKA.Keygen}(1^\lambda)$ *consist of a pair of public key encryption keys and a pair of digital signing keys. They are generated by users prior to joining a group, where* λ *denotes the security parameter. Public keys are used to invite parties to join a group.*

Initialize a Group**:** *Let* $\mathrm{G} = (\mathrm{ID}_1, \ldots, \mathrm{ID}_n)$*. For* $i \in [2, n]$ *let* pk_i *be an InitKey PK of party* ID_i*. Party* ID_1 *creates a new group with membership* G *by running:*

$$(\gamma, [W_2, \ldots, W_n]) \leftarrow \mathrm{CGKA.Init}\left(\mathrm{G}, [\mathrm{pk}_1, \ldots, \mathrm{pk}_n], [\mathrm{svk}_1, \ldots, \mathrm{svk}_n], (\mathrm{sk}_1, \mathrm{ssk}_1)\right)$$

which outputs a list of welcome messages W_i *for each party* ID_i*, and a local state for* ID_1*.*

Adding a Member**:** *A group member with local state* γ *can add party* ID *to the group by running* $(\gamma, W, M) \leftarrow \mathrm{CGKA.Add}(\gamma, \mathrm{ID}, (\mathrm{pk}, \mathrm{svk}))$*, which outputs a welcome message* W *for party* ID*, and an* add message M*, potentially update the state in the process.*

Removing a Member**:** *A group member with local state* γ *can remove group member* ID *by running* $(\gamma, M) \leftarrow \mathrm{CGKA.Rem}(\gamma, \mathrm{ID})$*, which outputs the* remove message M *and potentially updates the state in the process.*

Update**:** *A group member with local state* γ *can perform an update by running* $(\gamma, M) \leftarrow \mathrm{CGKA.Upd}(\gamma)$*, which outputs* update message M *and a (potentially) updated state.*

Collect and Deliver**:** *The delivery server, upon receiving a set of CGKA protocol messages* $M = (M_1, \ldots, M_k)$ *(including welcome messages) generated by a set of parties, sends out a round message* $(\gamma_{\mathrm{ser}}, (\hat{M}_1, \ldots, \hat{M}_n)) = \mathrm{CGKA.Dlv}(\gamma_{\mathrm{ser}}, M)$*, where* $\hat{T}_i$ *is the message for user* i *and* γ_{ser} *is the server's internal state.*

Process**:** *Upon receiving an incoming CGKA message* $\hat{M}_i$*, a party immediately processes it by running* $\gamma \leftarrow \mathrm{CGKA.Proc}(\gamma, \hat{M}_i)$*.*

GET GROUP KEY: *At any point a party can extract the current group key I from its local state γ by running* $I \leftarrow \mathrm{CGKA.Key}(\gamma)$.

2.4 Security Model for CGKA

Throughout this thesis, unless otherwise expressed, we anticipate an adversary who works in rounds, and that in each round may adaptively choose an action, including start/stop corrupting a party, instruct a party to initalize an operation, or relay a message.[1] The adversary can choose to corrupt any party, after which its state becomes fully visible to the adversary. In particular, corrupting a party gives the adversary access to the random coins used by said party when executing any group operation, deeming the party's actions deterministic in the eyes of the adversary throughout the corruption, which the adversary can choose when to stop. We would like to stress that security in this strong model implies security in weaker and potentially more realistic models, e.g. consider the setting where malware in a device leaks some of the randomness bits but cannot modify them. The adversary can also instruct a party to initalize an init/update/remove/add operation. This party then immediately outputs the corresponding message to be sent to the delivery server.

The goal of the adversary is to break the security of a target group key that was, at some point in the execution, considered to be the current group key by at least one group member, and that given the actions taken by the adversary so far is not trivially insecure. This means this key cannot be trivially decrypted from ciphertexts observed so far using secret keys leaked by corrupted parties. This can be ensured by defining predicates that specify for which group keys this is the case. Deciding whether this predicates holds can be determined by just looking at the transcripts (including corruption queries) of the individual group members and not some complicated global structure like the relative position of parties in the tree. Having such a simple predicate is important as we want a security notion which has a simple intuition behind it and in particular clearly captures FS and PCS.

The model will assume the existence of some *public key infrastructure* (PKI) that all users have access to. That is, a service that allows users to fetch encryption and signature keys from any other user in the network. Deploying such a system in practice is no easy task, particularly when we want to have the certainty that we are fetching correct and up-to-date keys. This falls outside the scope of this thesis and we refer the

[1] This section essentially replicates, with permission, parts of the full version [ACC+19] of our publication [KPPW+21]. The parts replicated here have been used in the thesis of co-author Karen Klein [Kle21]. These are not claimed as contributions of the present thesis, and rather are included here to serve an introduction to the framework used to prove security of many CGKA protocols and, in particular, to the security proofs that are part of Chapters 5 and 6.

reader to works tackling this for a centralized server, such as [MKKS+23, LCG+24], or the Key Transparency IETF working group [2]. On the more impractical side of things, one could also consider a decentralized system, in the style of PGP's *Wed of Trust*, which offloads key management to users.

Finally, only a single challenge per game is considered for simplicity; a standard hybrid argument allows one to extend security to multiple challenges, with a loss linear in the number of challenges (see, e.g., Lemma 6 in [ACDT20]).

Defintion 2.4.1 (Asynchronous CGKA Security)**.** *The security for CGKA*[3] *is modelled using a game between a challenger* C *and an adversary* A*. At the beginning of the game, the adversary queries* `create-group`(G) *and the challenger initialises the group* G *with identities* $(\mathrm{ID}_1, \ldots, \mathrm{ID}_\ell)$*. The adversary* A *can then make a sequence of queries, enumerated below, in any arbitrary order. On a high level,* `add-user` *and* `remove-user` *allow the adversary to control the structure of the group, whereas the queries* `confirm` *and* `process` *allow it to control the scheduling of the messages. The query* `update` *simulates the refreshing of a local state. Finally,* `start-corrupt` *and* `end-corrupt` *enable the adversary to corrupt the users for a time period. The entire state (old and pending) and random coins of a corrupted user are leaked to the adversary during this period.*

1. `add-user`$(\mathrm{ID}, \mathrm{ID}')$*: a user* ID *requests to add another user* ID' *to the group.*
2. `remove-user`$(\mathrm{ID}, \mathrm{ID}')$*: a user* ID *requests to remove another user* ID' *from the group.*
3. `update`(ID)*: the user* ID *requests to refresh its current local state* γ*.*
4. `confirm`(q, β)*: the* q*-th query in the game, which must be an action* $\mathrm{a} \in \{$`add-user`, `remove-user`, `update`$\}$ *by some user* ID*, is either confirmed (if* $\beta = 1$*) or rejected (if* $\beta = 0$*). In case the action is confirmed,* C *updates* ID*'s state and deletes the previous state; otherwise* ID *keeps its previous state).*
5. `process`(q, ID')*: if the* q*-th query is as above, this action forwards the (*W *or* T*) message to party* ID' *which immediately processes it.*
6. `start-corrupt`(ID)*: from now on the entire internal state and randomness of* ID *is leaked to the adversary.*

[2] `https://datatracker.ietf.org/wg/keytrans/about/`

[3] Here we reproduce the security model, introduced in [KPPW+21, Kle21] used to prove security of Tainted TreeKEM (the protocol introduced in Chapter 3). The security proofs in Chapters 5 and 6 are inspired in those of [KPPW+21, Kle21], and their models are very similar, though slightly adapted.

7. `end-corrupt`(ID)*: ends the leakage of user* ID*'s internal state and randomness to the adversary.*

8. `challenge`(q^*)*:* A *picks a query* q^* *corresponding to an action* $\mathrm{a}^* \in \{$`add-user`, `remove-user`, `update`$\}$ *or the initialization (if* $q^* = 0$*). Let* k_0 *denote the group key that is sampled during this operation and* k_1 *be a fresh random key. The challenger tosses a coin* b *and – if the safe predicate below is satisfied – the key* k_b *is given to the adversary (if the predicate is not satisfied the adversary gets nothing).*

At the end of the game, the adversary outputs a bit b' *and wins if* $b' = b$*. We call a CGKA scheme* (Q, ε, t)*-CGKA-secure if for any adversary* A *making at most* Q *queries of the form* `add-user`$(\cdot,\cdot)$*,* `remove-user`$(\cdot,\cdot)$*, or* `update`$(\cdot)$ *and running in time* t *it holds that*

$$\mathsf{Adv}_{\mathsf{CGKA}}(\mathsf{A}) := |\mathsf{Pr}[1 \leftarrow \mathsf{A}|b = 0] - \mathsf{Pr}[1 \leftarrow \mathsf{A}|b = 1]| < \varepsilon.$$

Additionally, one needs to define a *safe predicate* to rule out trivial winning strategies and, at the same time, restrict the adversary as little as possible. For example, if the adversary challenges the first (`create-group`) query and then corrupts a user in the group, it can trivially distinguish the real group key from random. Thus, intuitively, a query q^* is called *safe* if the group key generated in response to query q^* is not computable from any compromised state. Since each group key is encrypted to at most one key for each party, this means that the users which are group members[4] at time q^* must not be compromised as long as these keys are part of their state. However, defining a reasonable safe predicate in terms of allowed sequences of actions is very subtle.

To gain some intuition, consider the case where query q^* is an update for a party ID^*. Then, clearly, ID^* must not be compromised right after it generated the update. On the other hand, since the update function was introduced to heal a user's state and allow for PCS, any corruption of ID^* *before* q^* should not harm security. Similarly, any corruption of ID^* *after* a further processed update operation for ID^* should not help the adversary either (compare FS). Finally, also in the case where the update generated at time q^* is rejected to ID^* and ID^* processes this message of the form `confirm`$(q^*, 0)$ by returning to its previous state, any corruption of ID^* after processing the reject message should not affect security of the challenge group key. All these cases should be considered safe.

[4] To be precise, since parties might be in inconsistent states, group membership is not unique but rather depends on the users' *views* on the group state.

Additionally, the predicate has to take care of other users which are part of the group when the challenge key is generated: For a challenge to be safe, it must be that the challenge group key is not encrypted to any compromised key. At the same time, one has to be aware of the fact that in the asynchronous setting the view of different users might differ substantially. As mentioned above, the model considers inconsistency of user's states rather a matter of functionality than security, and aims to define the safe predicate as unrestrictive as possible, to also guarantee security for inconsistent group states. For example, consider the following scenario: user ID generates an update during an uncompromised time period and processes a reject for this update still in the uncompromised time period, but this update is confirmed to and processed by user ID^* before they do their challenge update q^*; this results in a safe challenge, since the challenge group key is only encrypted to the new init key, which is not part of ID's state at any compromised time point. However, one has to be careful here, since in a similar scenario where ID does not process the reject for their own update, the challenge group key would clearly not be safe anymore.

The following definitions consider discrete time steps measured in terms of the number of queries that have been issued by the adversary so far. We first identify for each user a critical window in the view of a specific user ID^*. The idea is to define exactly the time frame in which a user may leak a group key if ID^* generates it at a specific point in time and distributes it to the group. Clearly, the users may not be corrupted in this time frame if this happens to be the challenge group key.

Defintion 2.4.2 (Critical window, safe user)**.** *Let* ID *and* ID^* *be two (not necessarily different) users and* $q^* \in [Q]$ *be some query. Let* $q^- \leq q^*$ *be the query that set* ID*'s current key* in the view of ID^* at time q^**, i.e. the query* $q^- \leq q^*$ *that corresponds to the last update message* $\mathrm{a}^-_{\mathrm{ID}} := \mathtt{update}(\mathrm{ID})$ *processed by* ID^* *at some point* $[q^-, q^*]$ *(see Figure 2.1). If* ID^* *does not process such a query then we set* $q^- = 1$*, the first query. Analogously, let* $q^+ \geq q^-$ *be the first query that invalidates* ID*'s current key, i.e.* ID *processes one of the following two confirmations:*

1. $\mathtt{confirm}(\mathrm{a}^-_{\mathrm{ID}}, 0)$*, the rejection of action* $\mathrm{a}^-_{\mathrm{ID}}$*; or*
2. $\mathtt{confirm}(\mathrm{a}^+_{\mathrm{ID}}, 1)$*, the confirmation an update* $\mathrm{a}^+_{\mathrm{ID}} := \mathtt{update}(\mathrm{ID}) \neq \mathrm{a}^-_{\mathrm{ID}}$.

If ID *does not process any such query then we set* $q^+ = Q$*, the last query. We say that the window* $[q^-, q^+]$ *is* critical *for* ID *at time* q^* *in the view of* ID^**. Moreover, if the user* ID *is* not corrupted *at any time point in the critical window, we say that* ID *is* safe *at time* q^* *in the view of* ID^**.*

We are now ready to define when a *group key* should be considered *safe*. The group key is considered to be safe if all the users that ID^* considers to be in the group are

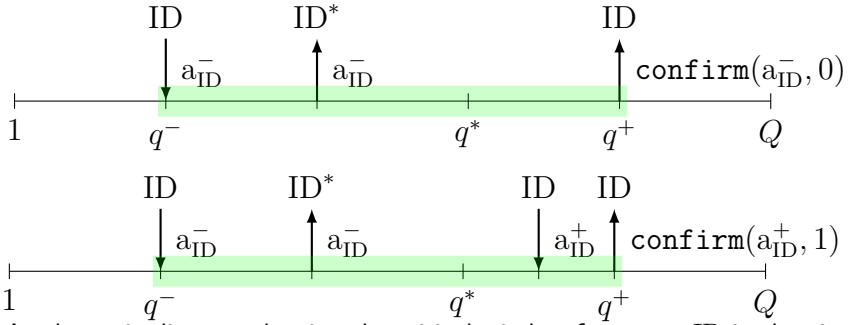

Figure 2.1: A schematic diagram showing the critical window for a user ID in the view of another user ID^* with respect to query q^*. An arrow from a user to the timeline is interpreted as a request by the user, whereas an arrow in the opposite direction is interpreted as the user processing the message. The figure at top (resp., bottom) corresponds to the first (resp., second) case in Definition 2.4.2.

individually safe, i.e., not corrupted in its critical window, in the view of ID^*. We point out that there is a exception when the action that generated the group key sk^* is a self-update by ID^* where, to *allow healing*, instead of the normal critical window we use the window $[q^*, q^+]$ as critical.

Defintion 2.4.3 (Safe predicate)**.** *Let* sk^* *be a group key generated in an action*

$$\mathrm{a}^* \in \{\texttt{add-user}(\mathrm{ID}^*, \cdot), \texttt{remove-user}(\mathrm{ID}^*, \cdot), \texttt{update}(\mathrm{ID}^*), \texttt{create-group}(\mathrm{ID}^*, \cdot)\}$$

at time point $q^* \in [Q]$ *and let* G^* *be the set of users which would end up in the group if query* q^* *was processed, as viewed by the generating user* ID^**. Then the key* sk^* *is considered* safe *if for all users* $\mathrm{ID} \in G^*$ *(including* ID^**) we have that* ID *is safe at time* q^* *in the view of* ID^* *(as per Definition 2.4.2) with the following exceptional case: if* $\mathrm{ID} = \mathrm{ID}^*$ *and* $\mathrm{a}^* = \texttt{update}(\mathrm{ID}^*)$ *then we require* ID^* *to be safe the window* $[q^*, q^+]$.

Looking ahead, in order to show security, the key step will be to show that if the safe predicate is satisfied for a group key Δ^* generated while playing the CGKA game, then none of the seeds or secret keys *used to derive* this group key are leaked to the adversary (Lemma 2.4.4). Given this security of the protocol can be argued as in [ACC+19] using the framework of Jafargholi et al. [JKK+17] (see the full paper for details). To this end, the general approach is to view the CGKA security game as a game on a graph and then define the *challenge graph* for challenge group key Δ^* as a sub-graph of the whole CGKA graph.

The CGKA graph. A node i in the CGKA graph for TTKEM is associated with seeds Δ_i and $s_i := H_2(\Delta_i)$ and a key-pair $(pk_i, sk_i) := \mathsf{Gen}(s_i)$. The edges of the graph, on the other hand, are induced by dependencies via the hash function H_1 or (public-key) encryptions. To be more precise, an edge (i, j) might correspond to either:

1. a ciphertext of the form $\mathsf{Enc}_{pk_i}(\Delta_j)$; or
2. an application of H_1 of the form $\Delta_j = H_1(\Delta_i)$ used in hierarchical derivation.

Naturally, the structure of the CGKA graph depends on the `update`, `add-user` or `remove-user` queries made by the adversary, and is therefore generated adaptively.

The challenge graph. The challenge graph for Δ^*, intuitively, is the sub-graph of the CGKA graph induced on the nodes from which Δ^* is trivially derivable. Therefore, according to the definition of the CGKA graph, this consists of nodes from which Δ^* is reachable and the corresponding edges (used to reach Δ^*). For instance, in the case where the adversary maintains all users in a consistent state the challenge graph would simply be the binary tree rooted at Δ^* with leaves corresponding to the leaf keys of users in the group at that point. When the group view is inconsistent among the users these leaves would correspond to the leaf keys of users in the view of ID^*. Below we state the key lemma which connects the safe predicate to the challenge graph of the corresponding CGKA protocol CGKA.

Lemma 2.4.4. *For any* safe *challenge group key in* CGKA *it holds that none of the seeds and secret keys in the challenge graph is leaked to the adversary via corruption.*

2.5 Ratchet Trees

Most CGKA constructions are inspired by TreeKEM and thus use the same underlying structure of a *ratchet tree*, a type of key-derivation graph, to manage keys. [5]

A ratchet tree is a directed left-balanced binary tree $\mathcal{T} = (V_{\mathcal{T}}, E_{\mathcal{T}})$, with edges pointing towards the root node v_{root}[6] and each user in the group associated to a leaf, i.e. a node without any children. We will use the notation $\mathcal{T}^i = (V^i_{\mathcal{T}}, E^i_{\mathcal{T}})$ to refer to the ratchet tree associated to round i. We say that a tree is *full* when all its number of leaves is a power of 2. When adding a new leaf to a full tree, a new root node is created, with said leaf and the previous root node being its parents.

Each node v is identified with a key-pair $(\mathrm{sk}_v, \mathrm{pk}_v)$ of a public-key encryption scheme. Roughly, each edge (w, v) corresponds to a ciphertext $\mathsf{Enc}_{\mathrm{pk}_w}(\mathrm{sk}_v)$[7] and each leaf node

[5] This Section replicates, with permission, parts of our publications [AAN+22b] and [AAB+21b].

[6] While many works in the literature consider edges going in the opposite direction, this non-standard direction of the edges better the hierarchical nature of secrets in the tree. In particular, the fact that knowledge of (the secret key associated to) the source node implies knowledge of (the secret key associated to) the sink node. Moreover, this definition of trees is much more intuitive in highlighting the connection between the protocols presented and the GSD game, which many CGKA security proofs use [KPPW+21]. Note that nodes therefore have one child and two parents.

[7] this is a simplification: in practice what is encrypted is a seed used later to generate the key-pair

v with a user u_v. A user u_v will know the (secret) key sk_v, and from the ciphertexts can then retrieve all the keys on the path from its leaf to the root v_{root}. The root key k is thus known to all users, and can be used for secure communication to or among the group members.

What makes this tree structure so appealing is the fact that in a group of size N, the key material of a user u can be completely rotated by replacing only the keys on the path from u to v_{root}, which in a balanced tree has length at most $d = \lceil \log(N) \rceil$. Moreover, as the nodes in a tree all have indegree two, one only needs to compute two fresh ciphertexts for each new key (in practice, as we will see, the new keys are derived via a hash-chain, so just one ciphertext per key will be needed).

The communication and computational efficiency of a key rotation are important aspects, as this is the main operation performed to add or remove users, or for a user to update their keys in order to recover from a potential compromise.

We will use the following notation:

- $\mathrm{child}(v)$ denotes the child of node v
- $\mathrm{parents}(v) = (\mathrm{lparent}(v), \mathrm{rparent}(v))$ denotes the tuple of v's left and right parent nodes.
- $\mathrm{partner}(v)$ denotes the *partner* of v, i.e., the parent of $\mathrm{child}(v)$ that is not equal to v.
- $\mathrm{path}(v, w)$ returns $\perp$ if w is not a descendant of v. Else, it returns v's path to w as $\mathrm{path}(v, w) = (v_0 = v, v_1, \ldots, v_k = w)$, where $v_i = \mathrm{child}(v_{i-1})$. We will often simply write $\mathrm{path}(v)$ for $\mathrm{path}(v, v_{root})$.
- $\text{co-path}(v)$ denotes v's co-path, i.e., the sequence of partners of nodes on $\mathrm{path}(v)$.
- $\mathrm{Int}(v, w)$ denotes their least common descendant, i.e., the node furthest away from the root in $\mathrm{path}(v) \cup \mathrm{path}(w)$. In a slight abuse of notation we will sometimes write $\mathrm{Int}(P, Q)$ for paths P and Q to denote the furthest node from the root in their intersection
- $\mathrm{isLeaf}(v)$ returns true if v is a leaf and false otherwise.

As hinted above, a node v in a ratchet tree have an associated *node state* $\gamma(v)$, containing a key-pair $(\mathrm{sk}, \mathrm{pk})$ from a PKE scheme, with the exception of the root node v_{root}, which simply contains an *update secret* k shared by all group members. Additionally, leaf nodes associated a user identifier ID, and a key-pair $(\mathrm{ssk}, \mathrm{svk})$ from a

signature scheme. Given a node v, we will refer to specific values in its state using . notation: e.g. $v.\mathrm{sk}$ to refer to its secret PKE key (for simplicity, particularly in Chapter 5 where notation is abundant, we will also sometimes just write sk_v). We will differentiate sometimes between the secret state of a node, containing the secrets keys sk and ssk (and possibly extra values), and the public state of a node ${}^p\gamma(v)$, containing the rest. Similarly, we will denote the collection of public states of nodes in a ratchet tree $\mathcal{T}$ by ${}^p\mathcal{T}$. While the public part of nodes' states can be accessed by all users, users should only have partial knowledge of the secret parts. Indeed, the protocol ensures that the secret part of $\gamma(v)$ is known only by users whose leaf is in the sub-tree rooted at v; this is known as the *tree invariant*.

Further we will define $\mathrm{leaf}(\mathrm{ID})$ to return the leaf node associated to user ID. Sometimes we will consider several trees containing the same (or similar subsets of) nodes, such as the different versions of a ratchet tree throughout protocol epochs. When needed, we will write $v^{\mathcal{T}}$ to clarify that we refer to node v in ratchet tree $\mathcal{T}$. Different protocols will additionally contain extra values in the nodes states, which will be defined in the relevant chapters.

Hierarchical derivation As mentioned above, a key update operation in protocols using ratchet trees is that by which users sample new keys along their leaves' paths. This operation of *hierarchical derivation* of keys is captured in the algorithm $\text{re-key}(v)$ (Figure 2.2). This methods can be traced back to TreeKEM, but is employed by most CGKA protocols (though sometimes with slight tweaks). Given a leaf v, the algorithm outputs a list of seeds and key-pairs for nodes along v's path. Here we define a more general version that can sample keys along incomplete paths, as this will be needed in Chapter 3. The algorithm uses two independent hash functions H_1 and H_2. These can be easily defined by taking a hash function H, fixing two different tags x_1 and x_2 and defining $\mathsf{H}_i(\cdot) = \mathsf{H}(\cdot, x_i)$.

Blank nodes and node resolution A node might be *blank*, meaning that its state is set to $\perp$. This will be the case for most nodes after group initialization and might also be the result of performing dynamic operations, or even certain key-updating operations, as we will see below. Each node in a ratchet tree will have an associated field $v.blank \in \{\mathsf{true}, \mathsf{false}\}$ denoting whether it is blank of not. In particular, no secret can be encrypted to a blank node, which brings us to the notion of a resolution. Intuitively, the resolution of a node v, $\mathrm{Res}(v)$, is the minimal set of non-blank nodes such that, for all leaves l in the subtree rooted at v, there is $w \in \mathrm{path}(l) \cup \mathrm{Res}(v)$. We can define it recursively as follows:

- If v is not blank, then $\mathrm{Res}(v) = v$

Algorithm re-key$(v, w; r)$

Require $w = \perp \vee \text{path}(v, w) \neq \perp$
If $w = \perp$**:** $w \leftarrow v_{root}$
$\Delta_0 \leftarrow_\$ \mathcal{S}(\lambda)$
$(\text{sk}_0, \text{pk}_0) \leftarrow \text{PKE.Gen}(\mathsf{H}_2(\Delta_0 \| r))$
$d \leftarrow |\text{path}(v, w)|$
For $i \in (1, \ldots, d-1)$**:**
  $\Delta_i = \mathsf{H}_1(\Delta_{i-1})$
  $(\text{sk}_i, \text{pk}_i) \leftarrow \text{PKE.Gen}(\mathsf{H}_2(\Delta_i))$
$\Delta_d \leftarrow \mathsf{H}_1(\Delta_{d-1})$
$\Delta \leftarrow (\Delta_1, \ldots, \Delta_d)$
If $w = v_{root}$**:**
  $\mathsf{K} \leftarrow \big((\text{sk}_1, \text{pk}_1), \ldots, (\text{sk}_{d-1}, \text{pk}_{d-1}), \Delta_d\big)$
Else:
  $(\text{sk}_d, \text{pk}_d) \leftarrow \text{PKE.Gen}(\mathsf{H}_2(\Delta_{d-1}))$
  $\mathsf{K} \leftarrow ((\text{sk}_1, \text{pk}_1), \ldots, (\text{sk}_d, \text{pk}_d))$
Return (Δ, K)

Figure 2.2: Algorithm re-key for node v and descendant w. It outputs a vector of hierarchically derived seeds, and another vector of corresponding keys for all nodes in v's path to w. λ is the security parameter, with $\mathcal{S}$ the seed space. It admits an additional optional fresh randomness. We will often write simply $\text{re-key}(v)$ for $\text{re-key}(v, v_{root})$

- If v is a blank leaf, then $\text{Res}(v) = \emptyset$
- Else, $\text{Res}(v) = \cup_{u \in \text{parents}(v)} \text{Res}(u)$

In an slight abuse of notation, we will use the term *co-path resolution* or Res_{co} to refer to the union of the resolutions of the nodes in the co-path of a node, i.e.

$$\text{Res}_{co}(v) = \cup_{w \in \text{co-path}(v)} \text{Res}(w)$$

.

User states Each group member should have a view of the public and secret information in the tree, namely keys, credentials, etc. that is consistent (though not overlapping since different users will know different secret information in order to preserve the tree invariant). More precisely, this is formalized by every user having an associated *protocol state* $\gamma(\text{ID})$ (or state for short when there is no ambiguity), which represents everything users need to know to stay part of the group, and which might satisfy certain properties (we implicitly assume a particular group, considering different groups secrets independent).

Different protocols will require this state to contain different fields, and thus the concrete definitions will be introduced in the relevant chapters. Nevertheless, states will typically include a copy of the ratchet tree, with the corresponding node states, a pending state γ' storing information from issued operations not yet processed, and additional values such as a hash of the protocol transcript, or MAC keys used to ensure/enhance robustness of the protocol. As with node states $\gamma.X$ will be used to refer to field X inside γ.

2.6 ART and TreeKEM

Asynchronous Ratcheting Tree (ART). The first proposal of (a simplified variant of) a CGKA is the Asynchronous Ratcheting Tree (ART) by Cohn-Gordon *et al.* [CCG+18].[8] This protocol uses ratchet trees where, recall, each party ID_i in the group is assigned their own leaf. In the case of ART, this is labelled with an ElGamal secret key x_i (known only to ID_i) and a corresponding public value g^{x_i}. The values of internal nodes are defined recursively: an internal node whose two parents have secret values a and b has the secret value g^{ab} and public value $g^{\iota(g^{ab})}$, where ι is a map to the integers. The secret value of the root is the group key. As illustrated in Figure 2.3, a party can update its secret key x to a new key x' by computing a new path from x' to the (new) root, and then sending the public values on this new path to everyone in the group so they can switch to the new tree. Note that the number of values that must be shared equals the depth of the tree, and thus (as the tree is balanced) is only logarithmic in the size of the group. A downside of ART is that, while possible, adding new users is not as practical as in other constructions. Indeed, the approach for doing so, as outlined in the original paper, requires the added user to become the partner of the user adding, by introducing a new node above the latter's leaf, and adding the former as the co-parent of it. This can easily create imbalances in the tree, with the alternative requiring synchronization between the parties.

The authors prove the ART protocol secure even against adaptive adversaries. However, in this case, their reduction loses a factor that is super-exponential in the group size. To get meaningful security guarantees based on this reduction requires a security parameter for the ElGamal scheme that is super-linear in the group size, resulting in large messages and defeating the whole purpose of using a tree structure.

TreeKEM. The TreeKEM proposal [BBR18], which is the CGKA underlying the MLS protocol [BBR+23], is similar to ART, as a group is still mapped to a ratchet tree where each node is assigned a public and secret value. In TreeKEM those values are the public/secret key pair for an arbitrary public-key encryption scheme. Unlike in

[8] This section replicates, with permission, parts of our publications [KPPW+21] and [AAB+21b].

ART, TreeKEM does not require any relation between the secret key of a node and the secret key of its parent nodes. Instead, an edge $u \to v$ in the tree (recall that edges are directed and pointing from the leaves to the root) denotes that the secret key of v is encrypted under the public key of u. This ciphertext can now be distributed to the subset of the group who knows the secret key of u to convey the secret key of v to them. We will refer to this as "encrypting v to u". Ever since its introduction on 2018, TreeKEM (through being part of MLS) has undergone a number of versions, at times with major changes between them, eventually leading to the current standard, published on July 2023. We will refer to version $X \in (1, \ldots, 20)$ of TreeKEM as TreeKEMvX. Below we will very roughly outline the protocol, along the way describing some of the changes happening between versions 7 to 9, which will be the most relevant for this thesis, as they together comprise most of the concepts used in both our protocols and in modern TreeKEM. More in detail, the transition from 7 to 8 is key, as it introduces the *Propose & Commit* (P&C) framework, which enables the execution of concurrent group operations. Version 9 introduces the notion of *unmerged leaves* in order to add parties. This is the way the current MLS standard handles additions and the main method used by related protocols, though not the only one, as we will see in Chapter 3. We will start introducing TreeKEMv7, together with the *unmerged leaves* technique, as it is conceptually simpler, to later outline the changes introduced by the P&C framework.

To *initialise a group*, the initiating party creates a ratchet tree, assigning its leaves to the keys of the invited parties. They then call algorithm $\mathrm{re\text{-}key}$ (Figure 2.2) on their path. That is, they sample fresh secret/public-key pairs for the nodes along their path and compute the ciphertexts corresponding to all the edges in the tree, i.e., they will encrypt the seed of each node to the keys of nodes in the resolution of its parent not in the initiator's path. (Note that leaves have no ingoing edges and thus the group creator only needs to know their public keys.) Finally they send all ciphertexts and public keys to the delivery server. If a party comes online, they receives from the server the public keys and the ciphertext corresponding to the node in the intersection between their path and that of the initiator and decrypts it (as it has the secret key of the leaf). This allows them to derive all the keys corresponding to nodes from said intersection all they way up to the root, including the group key.

To *update*, i.e., rotate their key material to achieve security against compromises, a user calls the $\mathrm{re\text{-}key}$ algorithm, generating new keys for the nodes in their path, together with the corresponding ciphertexts. These are then uploaded to the server, together with the new public keys. As with the group initialization, a user coming online can download the public keys and corresponding ciphertext and derive all the necessary new secret keys.

In order to *remove a party*, the remover simply issues a removal operation including the identifier of the removed party. When processing it, users will simply blank all the

nodes on the path of the removed user, effectively deleting all key material that was known to the removed user from the group state. Note that, in order to resume private communication, an update operation will need to follow in order to establish a new group key that the removed user does not have knowledge of. The concept of blanking and why it is needed will be discussed further in Chapter 3.

To *add a party* ID using the unmerged leaves technique, the adder ID' simply issues an add operation containing the public key and identifier of the added user, together with an update. The reason for the update is to ensure forward secrecy, i.e., that the added party only can only learn secrets generated after them joining the group. The added party will then be assigned an empty leaf on the tree, either the left-most free leaf, or a new one resulting from growing the tree. Then, ID' will send the corresponding keys and ciphertexts to the rest of the group as one would expect. As the new user will not have knowledge of intermediate keys on their path, ID' will encrypt the seed corresponding to the node in the intersection of their paths under the added party's leaf. When processing the message, group members will simply add the ID's identifier and key to the corresponding leaf, and then add the ID's identifier as *unmerged* to every node in ID's path from their leaf up to the node $\mathrm{Int}(\mathrm{ID}, \mathrm{ID}')$. A party being unmerged at a node signals that they do not have knowledge of the keys below it. Thus, when encrypting to a node, user's will also send additional encryptions under the public keys of the leaves of any unmerged parties. One can picture these as new leaves being connected directly to the node in the intersection between their path and that of the party adding them. As keys under said intersection get rotated, the party will learn them, thus becoming *merged*, not needing that extra personalized encryption.

Propose and Commit The Propose & Commit framework allows operations to be executed concurrently. Group operations are of two types: *proposals* and *commits*. The former can be either add, remove or update proposals and will not trigger any change in the group state when issued. Commits, on the other hand, will usher the group into a new epoch, generating a new group key while executing at the same time a set of proposals sent from the last epoch change.

More in detail, add and remove proposals will simply include the information of the added or removed party. Update proposals, in turn, will contain a new public key for the leaf of the updating party. When sending a commit, a user will sample new keys along their path by means of the $\mathrm{re\text{-}key}$ algorithm, and additionally collect all proposals received since they last processed a commit. When processing a commit, a user will update the keys along the committer's path, add or remove users as appropriate using the unmerged leaves technique and blanking, respectively, and update the leaf keys of all parties for which the commit contains an update proposal. Finally, they will blank all the nodes in the paths of said parties, up until the intersection with the path of the committer. This way, PCS can be achieved for any user that either proposed an update

or committed in that epoch, as any secret that was part of their state on the previous epoch gets either updated or erased.

It is clear that this approach allows for a more flexible and faster group evolution. In fact, it also allows for *external proposals*, coming from outside the group (e.g. the server suggesting a new user in the network be added, or a user that has not been online for a long time removed) to be committed by group members. The downside, however, comes from the introduction of blank nodes when committing update proposals, which can have a serious effect on efficiency, as we will discuss in Chapter 5.

Application Secrets. CGKA constructions employing key-derivation graphs like ratchet trees[9] rely on an additional mechanism to improve their forward-secrecy guarantees. Instead of directly using group keys k, associated with the root node, to communicate within the group, these keys are used to derive a so called *application secret* K that serves as the symmetric key for group communication. Whenever an update occurs, the new application secret of the group is computed as $K \leftarrow \mathrm{H}(k, K)$ the output of a hash function on input of the new group key and the previous application secret. Then, the old application secret is deleted from memory. The effect of this is that when a user's state leaks (including the current application secret K^t), no old application secret K^i can be recomputed from old update messages, unless K^{i-1} was already known to the adversary by former corruptions. In short, users gain the advantage of forward secrecy not only by issuing but also by processing updates of other users in the group. On top of the application secrets, other values serving different purposes are derived in a similar manner, making what is known as the *key-schedule*. We refer the reader to the standard [BBR+23] for more information.

[9] See Chapter 4 for a more general approach to building CGKA from key-derivation graphs that need not be ratchet trees.

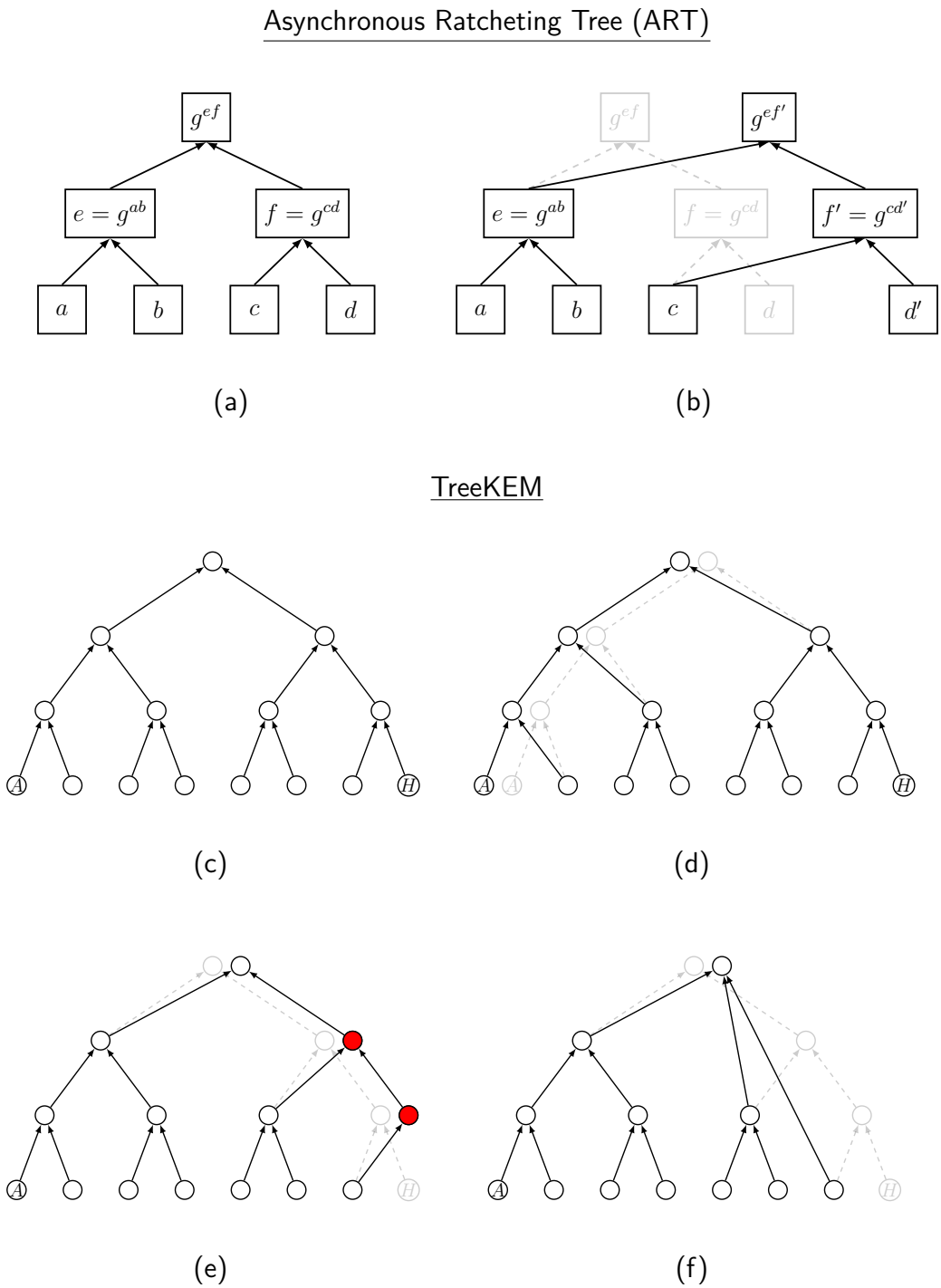

Figure 2.3: **Top**: Illustration of an update in the ART protocol. The state of the tree changes from (a) to (b) when Dave (node d) updates his internal state to d'. **Bottom**: update and remove in TreeKEM and TreeKEM with blanking. The state of a completely filled tree is shown in (c). The state changes from (c) to (d) when Alice (node A) performs an update operation. This changes to (e) when Alice removes Harry (node H) in naïve TreeKEM (with the nodes that Alice should not know in red) or to (f) in the actual TreeKEM protocol which uses blanking.

CHAPTER 3

Tainted TreeKEM

3.1 Introduction

In this Chapter[1] we introduce Tainted TreeKEM, a variant of TreeKEM characterized by an alternative method to remove and add parties, which we termed *tainting*. Let us start by recalling how membership changes in TreeKEM take place.

The ratchet tree structure of TreeKEM discussed in 2.6 and illustrated in Figure 2.3 would naïvely allow for adding and removing parties as follows. If ID_i wants to remove ID_j, they simply sample a completely fresh path from a (fresh) leaf to a (fresh) root replacing the path from ID_j's leaf to the root. They then compute and shares all the ciphertexts required for the parties to switch to this new path *except* the ciphertext that encrypts to ID_j's leaf. ID_i can add ID_j similarly, they just sample a fresh path starting at a currently not occupied leaf, using ID_j's key as the new leaf node, and communicates the new keys to ID_j. As mentioned before, this process can be optimized if the keys are derived hierarchically, from a hash chain of seeds, so that a single seed needs to be encrypted to each party.

Unfortunately, adding and removing parties like this creates a new problem. After ID_i added or removed ID_j, it knows all the secret keys on the new path (except the leaf). To see why this is a problem, assume ID_i is corrupted while adding (or removing) ID_j (and no other corruptions ever take place), and later – once the adversary loses access

[1] This Chapter essentially replicates, with permission, large parts of the full version [ACC+19] of our publication [KPPW+21]. Some parts of said paper, in particular those related to adaptive security, have been used in the thesis of co-author Karen Klein [Kle21]. There is no overlap in the technical results used in the two theses, just some parts of the introduction and informal protocol description have been used in both.

to ID_i's state – ID_i executes an update. Assume we use a naïve protocol where this update replaces all the keys on the path from ID_i's leaf to the root (as in ART) but nothing else. As ID_i's corruption also leaked keys not on this path, thus not replaced with the update, the adversary will potentially still be able to compute the new group key, so the update failed to achieve PCS.

To address this problem, TreeKEM introduced the concept of *blanking*, as mentioned in Section 2.6. In a nutshell, TreeKEM wants to maintain the invariant that parties know only the secrets for nodes on the path from their leaf to the root. However, if a party adds (or removes) another party as outlined above, this invariant no longer holds. To fix this, TreeKEM declares any nodes violating the invariant as not having any secret (nor public) value assigned to them. Such nodes are said to be *blanked*, and the protocol basically specifies to act as if the child of a blank node is connected directly to the blanked node's parents. In particular, when TreeKEM calls for encrypting something to a blank node, users will instead encrypt to this node's parents. In case one or both parents are blanked, one recurses and encrypts to their parents and so forth.

This saves the invariant, but hurts efficiency, as we now no longer consider a binary tree and, depending on the sequence of adds and removes, can end up with a "blanked" tree that has effective indegree linear in the number of parties, and thus future update, add or remove operations can require a linear number of ciphertexts to be sent. The reason one can still hope for TreeKEM's efficiency to not degrade too much and stay close to logarithmic in practice comes from the fact that blanked nodes can heal: whenever a party performs an update operation, all the blank nodes on the path from its leaf to the root become normal again.

The protocol studied in this paper builds closely on TreeKEM, particularly on versions pre-TreeKEMv8, where the P&C framework was introduced.

3.1.1 Our Contribution

In this chapter we formalize an alternative CGKA design, stemming from TreeKEM, first proposed by Millican on the MLS mailing list on February 2018[2], which we call Tainted TreeKEM, or simply TTKEM. Further, we discuss how its efficiency compares to that of TreeKEM and, in particular, show it to be more efficient on certain realistic scenarios. The full version [ACC+19] of the work on which this chapter is based on also proves that TTKEM satisfies a comprehensive security statement which captures the intuition that an update fixes a compromised state. This proof can be easily adapted to TreeKEM, for which we can get exactly the same security statement. This result

[2][MLS] Removing members from groups Jon Millican {jmillican@fb.com} 12 February 2018 https://mailarchive.ietf.org/arch/msg/mls/4-gvXpc-LGbWoUS7DKGYG65lkxs

was featured in Karen Klein's PhD thesis [Kle21] and is therefore not included in the main body, nor claimed as a contribution of this thesis.

Tainted TreeKEM (TTKEM) As just outlined, the reason TreeKEM can be inefficient comes from the fact that once a node is blanked, we cannot simply encrypt to it, but instead must encrypt to both its parents, if those are blanked, to their parents, and so forth. The rationale for blanking is to enforce an invariant which states that the secret key of any (non-blanked) node is only known to parties whose leaves are ancestors of this node. This seems overly paranoid, assume Alice removed Henry as illustrated in Figure 2.3, then the red nodes must be blanked as Alice knows their value, but it is instructive to analyze when this knowledge becomes an issue if no blanking takes place: If Alice is not corrupted when sending the remove operation to the delivery server there is no issue as she will delete secret keys she should not know right after sending the message. If Alice is corrupted then the adversary learns those secret keys. But even though now the invariant doesn't hold, it is not a security issue as an adversary who corrupted Alice will know the group key anyway. Only once Alice updates (by replacing the values on the path from her leaf to the root) there is a problem, as without blanking not all secret keys known by the adversary are replaced, and thus he will be able to decrypt the new group key; something an update should have prevented (more generally, we want the group key to be safe once all the parties whose state leaked have updated).

Keeping dirty nodes around, tainting versus blanking In TTKEM we use an alternative approach, where we do not blank nodes, but instead keep track of which secret keys of nodes have been created by parties who are not supposed to know them. Specifically, we refer to nodes whose secret keys were created by a party ID_i which is not an ancestor of the node as *tainted (by* ID_i*)*. The group keeps track of which nodes are tainted and by whom. A node tainted by ID_i will be treated like a regular node, except for the cases where ID_i performs an update or is removed, in which it must get updated.

Let us remark that tainted nodes can heal similarly to blanked nodes: once a party performs an update, all nodes on the path from its leaf to the root are no longer tainted.

Efficiency of TTKEM vs TreeKEM Efficiency-wise TreeKEM and TTKEM are incomparable. Depending on the sequence of operations performed either TreeKEM or TTKEM can be more efficient (or they can be identical). Thus, which one will be more efficient in practice will depend on the distribution of operation patterns we observe. In Section 3.3 we show that for some natural cases TTKEM will significantly outperform TreeKEM. This improvement is most patent in the case where a small subset of parties perform most of the add and remove operations. In practice, this could correspond

to a setting where we have a small group of administrators who are the only parties allowed to add/remove parties. The efficiency gap grows further if the administrators have a lower risk of compromise than other group members and thus can be required to update less frequently. In this setting, TTKEM approaches the efficiency of naïve TreeKEM.

When we compare the efficiency of the CGKA protocols we focus on the upload cost, i.e. the number of ciphertexts a party must exchange with the delivery server when issuing an (update, add or remove) operation.

3.1.2 Related work

Beyond the related work already mentioned in Section 1.3, we highlight here some results that are particularly relevant in the context of this Chapter.

To begin, it should be noted that, although current versions of MLS use the P&C framework and thus differ substantially from our protocol, TTKEM can also be easily cast in that same fashion, with commiters sampling (tainted) keys for all the nodes in the paths of users who submitted an update proposal. This is indeed done in [ACJM20]. As with TreeKEM, the application of this framework would bring an efficiency tradeoff that should be studied carefully and which we consider an interesting open question, though noting the challenge in doing so without real world data.

Even though the impact on subsequent operations of removals and adds have in TreeKEM and Tainted TreeKEM is different (see Section 3.3), the communication cost of the actual operations is the same, namely $\mathcal{O}(d\log(n/d))$ to remove or add d users (if done concurrently). A recent paper by Anastos *et al.* [AAB+24b] shows that this is actually optimal for CGKA (or Multicast Encryption) protocols built from PRGs, PRFs, dual PRFs, secret sharing, and standard or updatable PKE (symmetric encryption, respectively).

Additionally, the work of Bienstock *et al.* [BDR20] proposes an upper bound that the authors show to have optimal communication complexity when it comes to a set of users healing concurrently, though with respect to a weak security model. In this model, the adversary is passive and corruptions do not leak randomness, and thus nor any keys sampled by the parties not later stored in their states. This means users in the protocol can sample keys for nodes outside their path without the need to blank or taint those nodes to ensure security. Thus the protocol can be seen as an instantiation of TTKEM in the P&C framework but without the need for tainting due to the different security model.

Regarding security, Hofheinz, Kastner and Klein [HKK23] propose an alternative approach to showing adaptive security which directly applies to the TTKEM's security

proof from [KPPW+21], giving a reduction in the standard model achieving only polynomial loss (in the original work this loss was only possible in the ROM).

3.2 Description of Tainted TreeKEM

3.2.1 Asynchronous Continuous Group Key Agreement Syntax

In this Chapter we use the already introduced Definition 2.3.1 of CGKA. Nonetheless, said definition is more general that this protocol needs. In particular, Tainted TreeKEM assumes implicit authentication and thus nor $\mathrm{CGKA.Keygen}$ nor $\mathrm{CGKA.Init}$ will output or input any signing keys (see the discussion on partially active adversaries from Section 1.1.2 for why this can be a reasonable approach). Additionally, since no concurrency between issued group operations, nor server interaction is considered, algorithm $\mathrm{CGKA.Dlv}$ will simply input single messages, which are then output unchanged.

Finally, we remark that while the protocol allows any group member to add a new party to the group as well as remove any member from the group it is up to the higher level message protocol (or even higher level application) to decide if such an operation is indeed permitted, in which case clients can always simply choose to ignore the add/remove message. At the CGKA level, though, all such operations are possible.

3.2.2 Overview

The protocol uses a directed binary tree $\mathcal{T}$ as described in 2.5 as an underlying structure, with node states additionally containing a *tainter ID* $\mathrm{taintID}$.

Recall that to achieve FS and PCS, and to manage group membership, it is necessary to constantly renew the secret keys used in the protocol. We will do this through the group operations *update*, *remove* and *add*. We will use the term *refresh* to refer to the renewal of a particular (set of) key(s) (as opposed to the group operation). Each group operation will refresh a part of the tree, always including the root and thus resulting in a new group key which can be decrypted by all members of the current group. As explained earlier (see Sections 2.4), we will assume the existence of a PKI that users have access to and ensures consistency between the users' views of the initialization keys of other users. Accordingly, when adding a new user, the relevant algorithm will simply input a public key that is assumed to be correct and up-to-date.

We assume that a party will only process operations issued by parties that (at the time of issuing) shared the same view of the tree. This can easily be enforced by adding a (collision-resistant) hash of the operations processed so far [DV19, JMM19][3]. In

[3]For efficiency reasons one could use a Merkle-Damgård hash so that from the hash of a (potentially long) string M we can efficiently compute the hash of M concatenated with a new operation t.

particular, we will consider users having a protocol state $\gamma = (\mathrm{ID}, \mathcal{L}, \mathcal{T}, \mathcal{H}, \gamma')$, where:

- ID is the user's identifier
- $\mathcal{L}$ denotes the set of group members, i.e. ID's that are part of the group
- $\mathcal{T}$ denotes the current ratchet tree and such that, for each group member, their credential is associated to a leaf node.
- $\mathcal{H}$ denotes the hash of the group transcript so far, to ensure consistency.
- γ' stores the updated group state resulting from the last group operation issued by ID while they wait for confirmation.

Recall a user will generally not have knowledge of the secret keys associated to all tree nodes. However, if they add or remove parties, they will potentially gain knowledge of secret keys outside their path. We observe that this will not be a problem as long as we have a mechanism to keep track of those nodes and refresh them when necessary, and so towards this end we introduce the concept of tainting.

Tainting. Whenever party ID_i refreshes a node not lying on their path to the root, that node becomes *tainted* by ID_i. Whenever a node is tainted by a party ID_i, that party has potentially had knowledge of its current secret in the past. So, if ID_i was corrupted in the past, the secrecy of that value is considered compromised (even if ID_i deleted that value right away and is no longer compromised). Even worse, all values that were encrypted to that node are compromised too. We will assign a tainter ID $v.\mathrm{taintID}$ to all nodes. This can be empty, i.e. the node is untainted, or corresponds to a single party's ID, that who last generated this node's secret but is not supposed to know it. The tainter ID of a node is determined by the following simple *tainting rules*.

- After ID initialises, all internal nodes not on ID's path become tainted by ID.
- If ID updates or removes someone, refreshed nodes on ID's path become untainted.
- If ID updates or removes someone, all refreshed nodes *not* on ID's path become tainted by ID.

Hierarchical derivation of updates. We employ the same hierarchical derivation function $\text{re-key}(\cdot)$ as introduced in 2.5. It is worth noting that to prove security of the scheme in the standard model it suffices for the functions H_i to be pseudorandom functions (see Definition 2.2.2), with Δ_i the key and x_i the input.

With the introduction of tainting, however, it is no longer the case that all nodes to be refreshed lie on a path. Hence, we partition the set of all the nodes to be refreshed into paths and use a different seed for each path. For this we need a unique path cover, as users processing the update will need to know which nodes secrets depend on which. Formally, for a user ID, we want a set of paths $P_i = \{v_{i,0}, \ldots, v_{i,m_i}\}$ such that every tainted node is in some path P_i and moreover:

1. $\text{child}(v_{i,j}) = v_{i,j+1}$ for $j < m_i$ *(P_i is a path)*
2. $v_{i,j} \neq v_{k,l}$ if $i \neq k$ for any j, l *(each node is only in one path)*
3. $v_{i,0}.\text{taintID} = \text{ID}$ *(the start of each path is a node tainted by id)*
4. $\forall i, j : \text{child}(v_{i,m_i}) \neq v_{j,0}$ *(paths are maximal)*
5. $P_i \bigcap P_{id} = \emptyset$ *(paths are disjoint from main path to root)*
6. $\text{child}(v_{i,m_i}) \in P_{id} \vee \text{child}(v_{j,m_j}) \in P_i$ with $i < j$ *(the partition is unique)*
7. $v_{i,0} < v_{j,0}$ if $i < j$ *(there is a total ordering on paths)*

where P_{ID} is the path from the user's leaf to the root and $v_i < v_j$ if v_i is more to the left in a graphical representation of the tree (any total ordering on vertices suffices). We denote this ordered partition by $\text{tainted-paths}(\text{ID})$. Note that the first five conditions ensure that the partition contains only the nodes to be refreshed and that its size is minimal, while the sixth and seventh conditions guarantee that the partition is unique. A common ordering of the paths is needed, since when we refresh two paths that "intersect" (such that $\text{child}(v_{i,m_i}) \in P_j$, as the blue and red paths in the image below for example), the node secret in the "upper" path (the red path in this example) needs to be encrypted under the *new* public key of the node in the "lower" path (the new blue node) to achieve PCS. Thus, in this case, the blue path will need to be refreshed before the red one when processing the update. In general we will refresh paths right to left, i.e. P_i will be refreshed after P_j if $i < j$.

Let us stress that a party processing an update involving tainted nodes might need to retrieve and decrypt more than one encrypted seeds, as the refreshed nodes on its path might not all be derived hierarchically. Nonetheless, party needs to decrypt at most $\log n$ ciphertexts in the worst case.

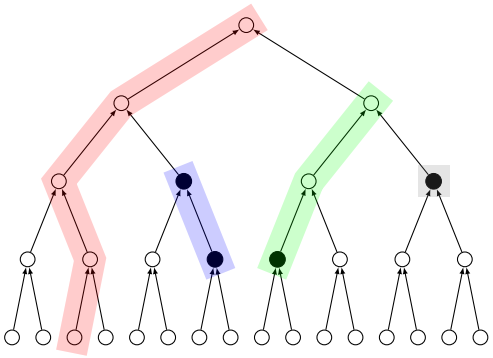

Figure 3.1: Path partition resulting from an update by Charlie (3^{rd} leaf node), with nodes tainted by him shown in black. To process it the grey node must be updated before the green path and the blue path before Charlie's (in red).

3.2.3 Tainted TreeKEM Operations

Whenever a user wants to perform a group operation, they will generate the appropriate update, add or remove message and send it to the delivery server, which will then either confirm or reject it. If the (honest) delivery server confirms an operation it will also deliver it to all the group members, who will process it and update their states accordingly. The initiator of a group operation creates a message M which contains all information needed by the other group members to process it (though different members might only need to retrieve a part of M for performing the update) and in case of an add also a welcome message W for the new member. A protocol message $M = (\mathrm{ID}, \mathsf{op}, C, \bar{\mathrm{pk}}, \mathcal{H})$ contains the following fields:

- $M.\mathrm{ID}$ - ID of the sender
- $M.\mathsf{op}$ - type of operation (remove/add/update), with potentially additional information about the added or removed user.
- $M.C$ - vector of ciphertexts which encrypt the seeds under the appropriate keys of all refreshed nodes
- $M.\bar{\mathrm{pk}}$ - vector of new public keys (derived from the new seeds) for all refreshed nodes
- $M.\mathcal{H}$ - hash-transcript

A welcome message $W = (\mathrm{ID}, \texttt{wel}, c, \mathcal{T}, \mathcal{L}, \mathcal{H})$ would also contain the type of operation (`wel`) and the sender ID, but additionally include:[4]

[4] In a functional group messaging protocol, a new member should also be communicated the current symmetric epoch key used to communicate text messages. As this is not strictly part of the

- $W.c$ - an encryption of the child node's seed
- $W.\mathcal{T}$ - the current tree structure, with public keys
- $W.\mathcal{L}$ - current list of group members
- $W.\mathcal{H}$ - current hash-transcript of the group

We describe the protocol with the help of two helper algorithms refresh and proc-refresh, which are defined in Figure 3.6. The former will take as input the state of the issuing user, as well as a user identifier ID, whose keys are to be refreshed (the same as the issuing user in the case of an update, a different one in the case of an add or remove), and will do as follows. It will generate new seeds for all nodes on ID's path, as well as all nodes tainted by ID, compute all the corresponding encryptions and keys for these nodes, store the resulting tree in the pending state γ', and output the updated state together with the ciphertexts and new public keys. As its counterpart, the former algorithm will take as input a user's state, a list of ciphertexts and public keys, and the identifiers of both the updated path's user and the issuer. It will decrypt the appropriate ciphertexts and update the ratchet tree with the obtained seeds and received public keys, outputting the resulting state.

Here and in the protocol description (Figure 3.5) we use the following helper functions:

- $\text{recover-ctxt}(C, v)$ returns the ciphertext from the list of ciphertexts C that is encrypted under the public key of v.
- $\text{recover-pk}(\bar{\text{pk}}, v)$ returns the new public key from $\bar{\text{pk}}$ that corresponds to v.
- $\text{add-party}(\mathcal{T}, \mathrm{ID})$ assigns the identifier ID to the left-most empty leaf of $\mathcal{T}$, growing the tree if it is full.

Below we provide intuitive overviews of each algorithm together with pseudocode in Figure 3.5. The description is accompanied by Figures 3.2, 3.3, and 3.4, showing examples of a update, remove and add operations, respectively. Note that, for the sake of simplicity, we assume that all received messages are well-formed and honestly generated. As a result, we do not, e.g., check that the public keys derived from a decrypted seed match those sent in the clear.

GCKA we ignore it for simplicity.

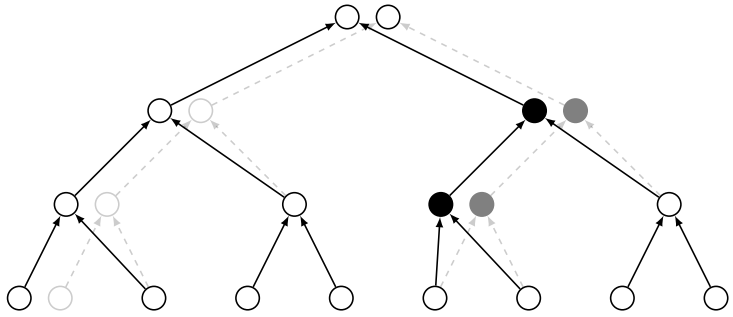

Figure 3.2: Example of an update operation by Alice (left-most leaf), who had tainted nodes (filled) as a result of, e.g., adding a party to the 5^{th} leaf. The state of the tree before the update is in a lighter shade.

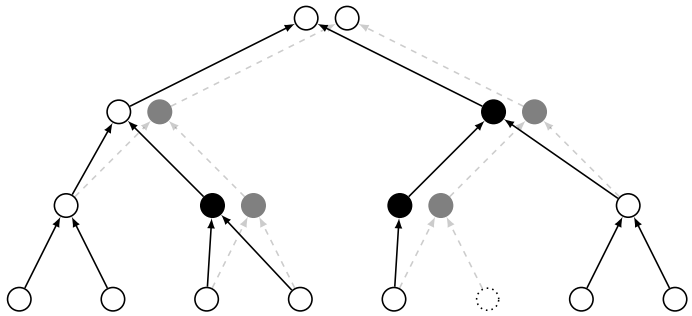

Figure 3.3: Example of a remove operation: Alice (left-most leaf) removes Frank (dotted) and in the process has to update his tainted nodes (filled). Old state is again showed in gray. Note that a node that was tainted by Frank is now untainted, as it lies on Alice's path.

Initialize. To create a new group with parties $\{\mathrm{ID}_1, \ldots, \mathrm{ID}_n\}$, a user ID_1 generates a new tree where the leaves correspond to the parties of the group (including themselves), with associated public keys the ones used to add them. The group creator then samples new key pairs for all the other nodes in the tree (optimizing with hierarchical derivation) and crafts welcome messages for each party.

Update. To perform an update, a user refreshes the nodes in its path to the root and also all the nodes tainted by him. We do this using the function refresh. It outputs the corresponding ciphertexts and new public keys.

Remove. To remove a user j, user i performs an update on behalf of j, refreshing all the nodes in j's path to the root as well as all nodes tainted by j (which will now become tainted by i). As with updates, we do this by calling the function refresh, adding information about the type of operation and the initiator of that operation. Note that a user cannot remove itself. Instead, we imagine a user that wants to leave the group could request for someone to remove her and delete her state.

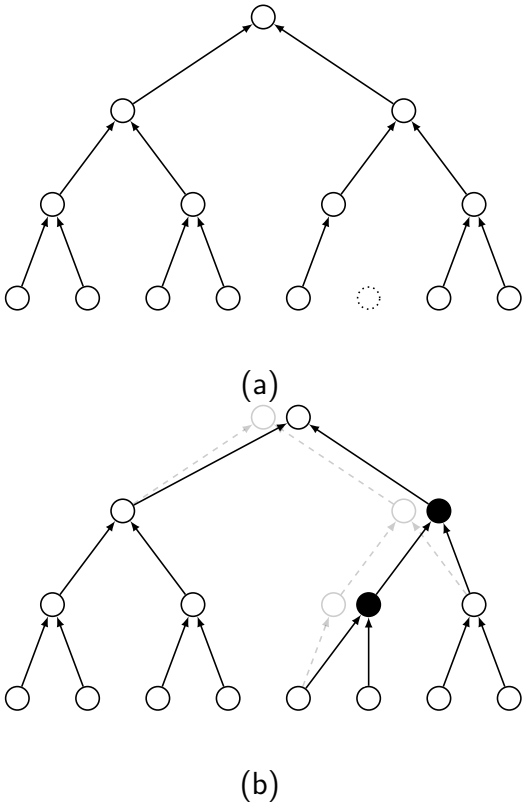

Figure 3.4: Example of an add operation. (a) illustrates the state of the tree before Alice adds Frank (6^{th} node), after which it turns into (b).

Add. To add a new member to the group, Alice identifies a free spot for them (for consistency, the left-most free spot), hashes her secret key together with some freshly sampled randomness to obtain a seed Δ, and derives seeds for the path to the root, overwriting the previous ones. They then encrypt the new seeds to all the nodes in the path (one ciphertext per node suffices given the hierarchical derivation). The reason for such a derivation of Δ is that the new keys will be secure against an adversary that does not have either knowledge of Alice's secret key or control/knowledge of the randomness used. We use h to refer to the hash function used.

Process. When a user receives a protocol message M, it identifies which kind of message it is and performs the appropriate update of their state. If it is a `confirm` or a `reject` it updates the current local state accordingly and remove the information in the pending local state.

Algorithm add$(\gamma, \mathrm{ID}', \mathrm{pk}')$

$(\mathrm{ID}, \mathcal{L}, \mathcal{T}, \mathcal{H}, \gamma') \leftarrow \gamma$
Require $\mathrm{ID} \neq \mathrm{ID}'$
$\gamma' \leftarrow (\mathrm{ID}, \mathcal{L}, \mathcal{T}, \mathcal{H}, \bot)$
$\gamma'.\mathcal{T} \leftarrow \text{add-party}(\gamma'.\mathcal{T}, \mathrm{ID}')$
$(\gamma', C, \bar{\mathrm{pk}}) \leftarrow \text{refresh}(\gamma', \mathrm{ID}')$
$(\text{leaf}(\mathrm{ID}')^{\gamma'.\mathcal{T}}.\mathrm{pk} \leftarrow \mathrm{pk}'$
$\gamma'.\mathcal{L} \leftarrow \gamma'.\mathcal{L} \cup \{\mathrm{ID}'\}$
$\gamma'.\mathcal{H} \leftarrow \mathsf{H}(\gamma'.\mathcal{H} \| (\mathrm{ID}, (\mathtt{add}, \mathrm{ID}', \mathrm{pk}'), C, \bar{\mathrm{pk}}))$
$\gamma \leftarrow (\mathrm{ID}, \mathcal{L}, \mathcal{T}, \mathcal{H}, \gamma')$
$M \leftarrow (\mathrm{ID}, (\mathtt{add}, \mathrm{ID}', \mathrm{pk}), C, \bar{\mathrm{pk}}, \gamma'.\mathcal{H})$
$W \leftarrow (\mathrm{ID}, \mathtt{wel}, \gamma'.\mathcal{L}, C[0], \gamma'.{}^{p}\mathcal{T}, \gamma'.\mathcal{H})$
Return (γ, M, W)

Algorithm remove(γ, ID')

$(\mathrm{ID}, \mathcal{L}, \mathcal{T}, \mathcal{H}, \gamma') \leftarrow \gamma$
Require $\mathrm{ID} \neq \mathrm{ID}'$
$\gamma' \leftarrow (\mathrm{ID}, \mathcal{L}, \mathcal{T}, \mathcal{H}, \bot)$
$(\gamma', C, \bar{\mathrm{pk}}) \leftarrow \text{refresh}(\gamma', \mathrm{ID}')$
$\gamma'.\mathcal{H} \leftarrow \mathsf{H}(\gamma'.\mathcal{H} \| (\mathrm{ID}, (\mathtt{rem}, \mathrm{ID}'), C, \bar{\mathrm{pk}}))$
$\gamma'.\mathcal{L} \leftarrow \gamma'.\mathcal{L} \setminus \{\mathrm{ID}'\}$
$\gamma \leftarrow (\mathrm{ID}, \mathcal{L}, \mathcal{T}, \mathcal{H}, \gamma')$
$M \leftarrow (\mathrm{ID}, (\mathtt{rem}, \mathrm{ID}'), C, \bar{\mathrm{pk}}, \gamma'.\mathcal{H})$
Return (γ', M)

Algorithm upd(γ)

$(\mathrm{ID}, \mathcal{L}, \mathcal{T}, \mathcal{H}, \gamma') \leftarrow \gamma$
$\gamma' \leftarrow (\mathrm{ID}, \mathcal{L}, \mathcal{T}, \mathcal{H}, \bot)$
$(\gamma', C, \bar{\mathrm{pk}}) \leftarrow \text{refresh}(\gamma', \mathrm{ID})$
$\gamma'.\mathcal{H} \leftarrow \mathsf{H}(\gamma'.\mathcal{H} \| (\mathrm{ID}, \mathtt{upd}, C, \bar{\mathrm{pk}}))$
$\gamma \leftarrow (\mathrm{ID}, \mathcal{L}, \mathcal{T}, \mathcal{H}, \gamma')$
$M \leftarrow (\mathrm{ID}, \mathtt{upd}, C, \bar{\mathrm{pk}}, \gamma'.\mathcal{H})$
Return (γ, M)

Algorithm process(γ, M)

If $M[1] = \mathtt{wel} \wedge \gamma = (\mathrm{ID}, (\mathrm{sk}, \mathrm{pk}))$**:**
  $(\mathrm{ID}', \mathtt{wel}, \mathcal{L}, c, \mathcal{T}, \mathcal{H}) \leftarrow M$
  $\gamma \leftarrow (\mathrm{ID}, \mathcal{L}, \mathcal{T}, \mathcal{H}, \bot)$
  $\gamma \leftarrow \text{proc-refresh}(\gamma, \mathsf{c}, \bot, \mathrm{ID}, \mathrm{ID}')$
  $(\text{leaf}(\mathrm{ID}).\mathrm{sk}, \text{leaf}(\mathrm{ID}).\mathrm{pk}) \leftarrow (\mathrm{sk}, \mathrm{pk})$
Elseif $(\mathrm{ID}, \mathcal{L}, \mathcal{T}, \mathcal{H}, \gamma') \leftarrow \gamma$**:**
  $(\mathrm{ID}', \mathtt{op}, C, \bar{\mathrm{pk}}, h) \leftarrow M$
  Require $h = \mathcal{H}$
  If $\mathtt{op} = \mathtt{upd}$**:**
    $\gamma \leftarrow \text{proc-refresh}(\gamma, \mathsf{C}, \bar{\mathrm{pk}}, \mathrm{ID}', \mathrm{ID}')$
  If $\mathtt{op} = (\mathtt{rem}, \mathrm{ID}_r)$**:**
  If $\mathrm{ID}' = \mathrm{ID}$**:**
    $\gamma \leftarrow \bot$
  Else:
    $\gamma \leftarrow \text{proc-refresh}(\gamma, \mathsf{C}, \bar{\mathrm{pk}}, \mathrm{ID}_\mathsf{r}, \mathrm{ID}')$
    $\gamma.\mathcal{L} \leftarrow \gamma.\mathcal{L} \setminus \{\mathrm{ID}'\}$
  If $\mathtt{op} = (\mathtt{add}, \mathrm{ID}_a, \mathrm{pk}) \wedge \mathrm{ID}_a \neq \mathrm{ID}$**:**
    $\gamma.\mathcal{T} \leftarrow \text{add-party}(\mathcal{T}, \mathrm{ID}_a)$
    $\gamma \leftarrow \text{proc-refresh}(\gamma, \mathsf{C}, \bar{\mathrm{pk}}, \mathrm{ID}_\mathsf{a}, \mathrm{ID}')$
    $\text{leaf}(\mathrm{ID}_a).\mathrm{pk} \leftarrow \mathrm{pk}$
    $\gamma.\mathcal{L} \leftarrow \gamma.\mathcal{L} \cup \{\mathrm{ID}_a\}$
    If $\mathtt{op} = \mathtt{confirm}$**:**
      $\gamma \leftarrow \gamma'$**;** $\gamma' \leftarrow \bot$
    ElseIf $\mathtt{op} = \mathtt{reject}$**:**
      $\gamma' \leftarrow \bot$
    Else:
  $\gamma.\mathcal{H} \leftarrow \mathsf{H}(\gamma.\mathcal{H} \| M)$
Return $(\gamma, \text{CGKA.Key}(\gamma))$

Figure 3.5: Tainted TreeKEM algorithms

Algorithm refresh(γ, ID')

00 $(\mathrm{ID}, \mathcal{L}, \mathcal{T}, \mathcal{H}, \gamma') \leftarrow \gamma$
01 $P_0 \leftarrow \mathrm{path}(\mathrm{leaf}(\mathrm{ID}'))$
02 $\gamma' \leftarrow (\mathrm{ID}, \mathcal{L}, \mathcal{T}, \mathcal{H}, \bot)$
03 $C \leftarrow \emptyset$
04 $\bar{\mathrm{pk}} \leftarrow \emptyset$
$\backslash\backslash$ **refresh all paths from tainted nodes to root**
05 $\{P_1, \ldots, P_n\} \leftarrow \text{tainted-paths}(\mathrm{ID}')$
06 **For** $i = n, \ldots, 0$**:**
07 $v_{i,0}, \ldots, v_{i,m} \leftarrow P_i$
08 $(\Delta, K) \leftarrow \text{re-key}(v_{i,0}, v_{i,m})$
09 $\backslash\backslash$ **encrypt first to parents of lowest node**
10 **For** $p \in \mathrm{parents}(v_{i,0})$**:**
11 $C \xleftarrow{\cup} \mathrm{PKE.Enc}(p.\mathrm{pk}, \Delta[0])$
12 **If** $v_{i,0} \in \mathrm{path}(\mathrm{leaf}(\mathrm{ID}))$**:**
13 $(v_{i,0}^{\gamma'.\mathcal{T}}.\mathrm{sk}, v_{i,0}^{\gamma'.\mathcal{T}}.\mathrm{pk}) \leftarrow K[0]$
14 **Else:** $v_{i,0}^{\gamma'.\mathcal{T}}.\mathrm{pk} \leftarrow K[0][1]$
15 $\bar{pk} \xleftarrow{\cup} K[0][1]$
16 $v_{i,j}^{\gamma'.\mathcal{T}}.\mathrm{taintID} \leftarrow \mathrm{ID}$
17 **For** $j = 1, \ldots, m$**:**
18 **If** $v_{i,j} \in \mathrm{path}(\mathrm{leaf}(\mathrm{ID}))$**:**
19 $(v_{i,j}^{\gamma'.\mathcal{T}}.\mathrm{sk}, v_{i,j}^{\gamma'.\mathcal{T}}.\mathrm{pk}) \leftarrow K[j]$
20 **Else:**
21 $v_{i,j}^{\gamma'.\mathcal{T}}.\mathrm{pk} \leftarrow K[j][1]$
22 $\bar{pk} \xleftarrow{\cup} K[j][1]$
23 $v_{i,j}^{\gamma'.\mathcal{T}}.\mathrm{taintID} \leftarrow \mathrm{ID}$
24 $\gamma \leftarrow (\mathrm{ID}, \mathcal{L}, \mathcal{T}, \mathcal{H}, \gamma')$
25 **Return** $(\gamma, C, \bar{\mathrm{pk}})$

Algorithm proc-refresh$(\gamma, \mathsf{C}, \bar{\mathrm{pk}}, \mathrm{ID}', \mathrm{ID}_\mathsf{t})$

26 $(\mathrm{ID}, \mathcal{L}, \mathcal{T}, \mathcal{H}, \gamma') \leftarrow \gamma$
27 $P_0 \leftarrow \mathrm{path}(\mathrm{leaf}(\mathrm{ID}'))$
28 $\{P_1, \ldots, P_n\} \leftarrow \gamma.\text{tainted-paths}(\mathrm{ID}')$
29 $\backslash\backslash$ **update all paths of tainted nodes**
30 **For** $i = n, \ldots, 0$**:**
31 $v \leftarrow \mathrm{Int}(P_i, \mathrm{path}(\mathrm{leaf}(\mathrm{ID})))$
32 **If** $v \neq \bot$**:**
33 $p \leftarrow \mathrm{parents}(v_i) \cap \mathrm{path}(\mathrm{leaf}(\mathrm{ID}))$
34 $c \leftarrow \text{recover-ctxt}(C, p)$
35 $\Delta \leftarrow \mathrm{PKE.Dec}(p.\mathrm{sk}, c)$
36 $(v_{i,0}, \ldots, v_{i,j_i}) \leftarrow P_i$
37 **For** $j \in (0, \ldots, j_i)$**:**
38 **If** $v_{i,j} \in \mathrm{path}(\mathrm{leaf}(\mathrm{ID}))$**:**
39 $(\mathrm{sk}, \mathrm{pk}) \leftarrow \mathrm{PKE.Gen}(\mathsf{H}_2(\Delta))$
40 $(v_{i,j}.\mathrm{sk}, v_{i,j}.\mathrm{pk}) \leftarrow (\mathrm{sk}, \mathrm{pk})$
41 $\Delta \leftarrow \mathsf{H}_1(\Delta)$
42 **Else:**
43 $v_{i,j}.\mathrm{pk} \leftarrow \text{recover-pk}(\bar{\mathrm{pk}}, v_{i,j})$
44 $v_{i,j}.\mathrm{taintID} \leftarrow \mathrm{ID}_t$
45 **Return** γ

Figure 3.6: Helper algorithms. All updates values for node states take place in the copy of the ratchet tree stored in the pending state γ'. It will only be when executing process that those changes will make it to the ratchet tree $\gamma.\mathcal{T}$.

3.3 Tainting versus Blanking

In terms of security there is little difference between what is achieved using tainting and using blanking. Updates have the same function: they refresh all known secrets, allowing for FS and PCS through essentially the same mechanism in both approaches. However, as mentioned before, tainting seems to be a more natural approach: it maintains the desired tree structure, and its bookkeeping method gives us a more complete intuition of the security of the tree. It also corresponds to a more flexible framework: since blanking simply forbids parties to know secrets outside of their path, it leaves little flexibility for how to handle the initialization phase.

With regards to efficiency, the picture is more complicated. TTKEM and TreeKEM[5] are incomparable in the sense that there exist sequences of operations where either one or the other is more efficient. Thus, which one is to be preferred depends on the distribution of operation sequences. We observe that there are two major differences in how blank and tainted nodes affect efficiency. The first one is in the set of affected users: a blank node degrades the efficiency of *any user* whose copath contains the blank. Conversely, a tainted node affects only *one* user; that who tainted it, but on the down side, it does so no matter where in the tree this tainted node is. The second difference is the healing time: to "unblank" a node v it suffices that some user assigned to a leaf in the tree rooted at v refreshes it (thereby overwriting the blank with a fresh key). However, to "untaint" v, simply overwriting it this way is necessary but not sufficient. In addition, it must also hold that no other node in the tree rooted at v is tainted.

Thus, intuitively, in settings where the tendency is for adds and remove operations (i.e. those that produce blanks or taintings) to be performed by a small subset of group members it is more efficient to use the tainting approach. Indeed, only update operations done by that subset of users will have a higher cost. As mentioned in the introduction, such a setting can arise quite naturally in practice – e.g. when group membership is managed by a small number of administrators.

To test this, we ran simulations comparing the number of ciphertexts (cost) users need to compute on average as a consequence of performing updates, adds and removes. Ideally, we would like to sample a sequence of group operations, execute them in both protocols and compare the total cost. However, this seems infeasible: in TreeKEM operations are collected into Commits, whereas in TTKEM these are applied one by one, separately. Hence, we compared TTKEM (referred to as *tainted* in the graphs) against two different simplified versions of TreeKEM, between which real TreeKEM lies efficiency-wise. The first version (*TKEM*), more efficient than actual TreeKEM, ignores Commits and just executes operations one by one, without the update that

[5]We compare TTKEM with the most recent version TreeKEMv9.

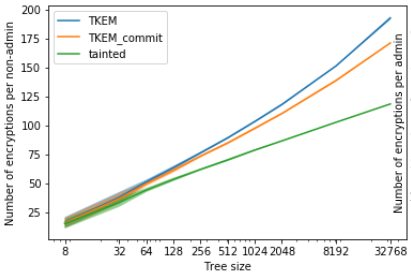

Figure 3.7: Cost for non-administrators

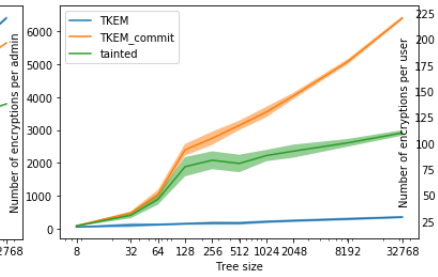

Figure 3.8: Cost for administrators

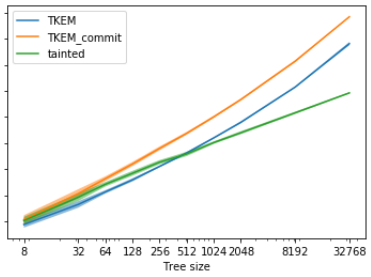

Figure 3.9: Average cost per user

would follow every Commit. The second version (*TKEM_commit*), less efficient than the real TreeKEM, enforces that every operation is committed separately, essentially performing an extra update operation after every add or remove.

We simulated groups of sizes between 2^3 and 2^{15} members. Trees of size 2^i were initialized with 2^{i-1} members and sequences of $10 * 2^i$ update/remove/add operations were sampled according to a $8:1:1$ ratio. One would expect for many more updates than add/removes to take place; but also, the more common updates are, the closer that efficiency is going to be to that of naïve TreeKEM for both TreeKEM and TTKEM. Thus, this seems a reasonable ratio that also highlights the differences between the protocols - it is also the ratio used by previous simulations used to analyze the efficiency of blanking in TreeKEM[6]. We test two different scenarios. In the first one we limit the ability of adding and removing parties to a small subset of users, the administrators. In the second, we make no assumption on who performs adds and removes and sample the authors of these uniformly at random.

To simulate the administrator setting (figures 3.7, 3.8 and 3.9), we set a small (1 per group in groups of size less tan 128 and 1 per every 64 users in bigger groups) random set of users to be administrators. Adds and removes are then performed by one of those administrators sampled uniformly at random. The removed users, as well as the authors of the updates were also sampled uniformly at random. Figures 3.7 and 3.8 illustrate that TTKEM allows for an interesting trade-off, where non-administrators enjoy more efficient updates at the expense of potentially more work for administrators. This would be favourable in settings where administrators have more bandwidth or computational power. When considering the average cost incurred by group member, admins or non-admins (figure 3.9), all three protocol perform similarly for smaller groups, with TTKEM behaving better asymptotically.

In the second scenario (figures 3.10 and 3.11), where adds and removes are performed

[6][MLS] Cost of the partial-tree approach. Richard Barnes {rlb@ipv.sx} 01 October 2018 `https://mailarchive.ietf.org/arch/msg/mls/hhl0q-OgnGUJS1djdmH1JBMqOSY/`

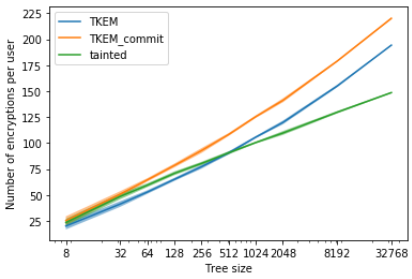

Figure 3.10: Updaters follow uniform dist.

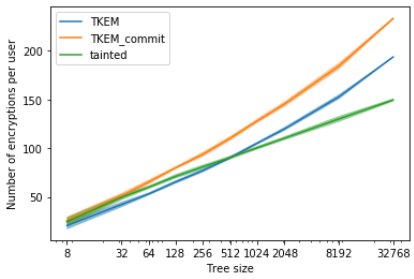

Figure 3.11: Updaters follow Zipf dist.

by users sampled uniformly, the results are similar: all protocols perform comparably on smaller groups, with TTKEM behaving more efficiently on larger groups. Here, we distinguish two further situations depending on the distribution on update authors (or *updaters* for short). Figure 3.10 shows the results of sampling updaters uniformly at random. This would reflect scenarios where updates are executed periodically, as in e.g. devices that are always online and where a higher level policy stipulates to update daily. In contrast, figure 3.11 shows the results of sampling updaters following a Zipf distribution. The Zipf distribution is used widely to model human activity in interactive settings. Recently, a study on messages sent on internet communities shows that the growth of messages sent per individual over time follows Gibrat's law [RBH+09]. This in turn implies that the distribution of the number of messages sent per individual at a point in time converges asymptotically to a Zipf distribution [Gab99]. Thus, the latter scenario models a setting where updates are correlated with the level of activity of the users, e.g. when the devices used are not always online.

Overall, while we cannot say TTKEM will be more efficient than TreeKEM in every setting, it is clear that it constitutes a promising CGKA candidate, which can bring efficiency improvements over TreeKEM in different realistic scenarios. Moreover, we would also like to point out that to improve the efficiency of these protocols, different policies can be implemented, such as strategically placing users on the tree: e.g. distributing administrators or frequent updaters closer to the right side of the tree, where more new users will be added.

In fact, the efficiency benefits of tainting can be further compounded if the users initiating add/remove operations also perform update operations less frequently than others. It turns out that this type of asymmetry between frequency of updates can arise through unrelated (yet realistic) reasons. Suppose, for example, we determine that Bob is at a significantly higher risk of compromise than Alice. Concretely, consider Alice – an admin – who works from the office and Bob, a non-IT professional who communicates using his cellphone in the field. On the one hand, Alice might (be instructed to) only use a well maintained and locked down high security device at the office while accessing the internet through a well defended corporate network. Conversely, Bob's mobile

device has a significantly higher risk of falling in the adversaries hands (at least briefly) compared to Alice's device that never leaves the office. Bob might access the internet through a variety of public and private networks. He may also be running a host of other apps on the same device, further raising his risk profile compared to Alice's. Finally, not being a trained IT professional like Alice, he might not be as proficient at preventing compromise; e.g. by detecting fishing attempts, avoiding dubious websites and apps and by using powerful but complicated defensive tools on the device. Under these and similar (quite realistic) types of conditions, it is reasonable to conclude that Bob's device is more likely to be compromised than Alice's, so it is rational to spend a larger proportion of the bandwidth dedicated to a given group chat session on updates for Bob than the bandwidth spent on Alice's updates. In particular, this better minimises the probability at any given point that the sessions privacy has been compromised when compared to (say) using the available bandwidth equally between the two.

Finally, since our simulations above are initialized with a fully healed tree (no blanks or taints), we discuss the different costs of the initialization phase, i.e. the process of creating a group and clearing all blanks or taints, in both protocols. We find that TTKEM can achieve such a healed tree considerably faster than TreeKEM.

Efficiency of Initialisation.

For many group chat sessions, the initialisation phase will be the most inefficient phase of the session's life-cycle, as inefficiencies arise by adding and removing members to a session. A group will certainly see at least as many adds as removes, and likely most of those adds will happen at the beginning of the group's life (either batched within **init** or just after it). Thus, the process by which a group goes from a initial state to a fully "healed" tree (that without blanks or taints) is of great importance. We will henceforth consider the scenario where a group is initialised with a large number of members and will study the cost (in particular, the number of ciphertexts) of transitioning to a fully healed ratchet tree in the least number of group operations (without further adds, removes or redundant updates).

While this is obviously quite a restrictive assumption, we believe it would be quite similar to the initial behaviour of most groups. In fact, a similar sequence of group operations could be somehow encouraged by a higher level protocol: a main aim for a group should be to achieve the ratcheting tree structure that gives log size packages for each operation as soon as possible.[7] We will assume that groups with blanking are initialised as fully blanked trees, except for the creator's path (To the best of our knowledge, mitigating double-joins via blanking does not allow for any other more

[7] In particular, the less bandwidth used per update the more updates can be performed and so the stronger the expected security properties for the session will be.

efficient initialisation procedure than this). We also recall that groups with tainting are initialised with a tree fully tainted by the initiator Alice.

In order to fully unblank (resp. untaint) the tree, we need every second member to update. In the tainted case, the order is irrelevant, as any update by a member other than the group creator involves $\log n$ ciphertexts, meaning we can achieve a fully untainted tree with a total communication cost of $(\lfloor n/2 \rfloor - 1) \log n$. This is not the case with blanking, where the order matters.

Lemma 3.3.1. *To transition from a fully blanked (except for the group creator's - the first leaf - path to root) tree to a fully unblanked tree, the following sequence of updates has minimal cost:*

$$n/2+1,\ n/4+1,\ 3n/4+1,\ n/8+1,\ 3n/8+1,\ 5n/8+1,\ 7n/8+1,\ n/16+1,\ \ldots$$

Proof. Let $\mathcal{T}_1$ denote the left subtree, and $\mathcal{T}_2$ the right subtree. If any user (with a leaf) in $\mathcal{T}_1$ updates before anyone in $\mathcal{T}_2$ does, $\mathcal{T}_2$ will be blank and hence one ciphertext per user in $\mathcal{T}_2$ will be needed. On the contrary, if some update from $\mathcal{T}_2$ has already taken place, all updates from $\mathcal{T}_1$ will just need one ciphertext to be communicated to $\mathcal{T}_2$, they will just need to encrypt the new group secret under the head of $\mathcal{T}_2$. Moreover, note that the cost of any update from $\mathcal{T}_2$ will be independent from the structure of $\mathcal{T}_1$, as, being on the group creator's path, the head of $\mathcal{T}_1$ will not be blank. Therefore, the optimal scenario is that someone from the right subtree updates first, assume its the user with the leaf in position $n/2+1$ without loss of generality. Following a similar argument, an update should then come from the right subtree of $\mathcal{T}_1$ before one from its left subtree (similarly for $\mathcal{T}_2$), and so on. □ □

Now, the cost (i.e. number of required ciphertexts) to update in this order is $(n/2-1)+1$ for the first member, $(n/4-1)+2$ for each of the two next ones, $(n/8-1)+3$ for the 4 next, and so on. We end up with the following lower-bound on the cost of healing:

$$\frac{n}{2} + 2\left(\frac{n}{4}+1\right) + 4\left(\frac{n}{8}+2\right) + \ldots + \frac{n}{4}\left(\frac{n}{n/2}+\log n - 2\right) = \tag{3.1}$$

$$= \frac{n}{2}(\log n - 1) + \sum_{i=1}^{\log n - 1} (i-1)2^{i-1} \tag{3.2}$$

$$= \frac{n}{2}(\log n - 1) + 2\left(2^{\log n - 2}(\log n - 2) - 2^{\log n - 2} + 1\right) \tag{3.3}$$

$$= n(\log n - 2) + 2 \tag{3.4}$$

Thus, even for the optimal update ordering, blanking is more costly by about a factor of 2.

CHAPTER 4

Multiple Groups

4.1 Introduction

In this Chapter[1] we consider an extension of the definition of ratchet tree from 2.5. We are given a base set $[N] = \{1, \ldots, N\}$ of users with a set system $\mathcal{S} = \{S_1, \ldots, S_k\}$ (each $S_i \subseteq [N]$), and we ask for a key managing structure such that for any set $S_i \in \mathcal{S}$, the users in S_i share a group key. This is a natural and well motivated setting; consider for example a university, where one might want to have a shared key for all students attending particular lectures; or an organization where members belong to (possibly several) different teams.

A trivial solution to this problem is to simply use a different key-tree for every group S_i. In this chapter we explore more efficient solutions.

Key-graphs beyond trees. For a set system $\mathcal{S}$ as above, instead of using disjoint trees, any directed acyclic graph (DAG) $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ with the following properties is sufficient to maintain group keys:

1. Every user $i \in [N]$ corresponds to a source v_i (a node of indegree 0).
2. Every group $S_i \in \mathcal{S}$ corresponds to a sink v_{S_i} (a node of outdegree 0).
3. For every $S_i \in \mathcal{S}$ and $j \in [N]$, there is a directed path from v_j to v_{S_i} if and only if $j \in S_i$.

[1] This Chapter essentially replicates, with permission, large parts of the full version [AAB+21a] of our publication [AAB+21b].

4. The indegree of any node is at most 2.

The first three properties ensure that any user $j \in [N]$ can learn the keys associated with the nodes of groups they are in. The last property is not really necessary, but it is without loss of generality in the sense that any graph can be turned into a graph with at most as large update cost (as we show in Section 4.3), and where every node other than the leaves has indegree at most 2. We call this a key-derivation graph for $\mathcal{S}$.

Update cost. If we rotate the keys of a user i we need to replace all keys that can be reached from v_i, which we denote by $\mathcal{D}(v_i)$, and encrypt each new key under the keys of its co-path. We thus define the update cost of a user $i \in [N]$ as $\sum_{v \in \mathcal{D}(v_i)}(\mathrm{indeg}(v)-1)$, which with item 4 above roughly simplifies to the number of v_i's descendants $|\mathcal{D}(v_i)|$. The update cost $\mathrm{Upd}(\mathcal{G})$ of a DAG $\mathcal{G}$ is the sum over the update cost of all its leaves, which is proportional to the average update cost of users.

Towards constructing more efficient key-derivation schemes when we have multiple overlapping groups, we thus address the problem of determining how small the update cost of a key-derivation for a given set system $\mathcal{S} = \{S_1, \ldots, S_k\}$ over $[N]$ can be, and how to find graphs which achieve, or at least come close to, this minimum.

Our contributions. We look at this problem from two perspectives. To get an insight on how much can be saved compared to the trivial solution, we first adapt a qualitative, asymptotic perspective, where we assume a fixed set system, but the number of users N goes to infinity while the relative size of the sets and intersections remains the same. We prove a lower bound on the update cost in this setting and give an algorithm computing graphs matching this bound.

As this solution turns out to be far from optimal for certain concrete set systems, we then also look at a quantitative non-asymptotic setting, where we consider concrete bounds and care about things like additive constants. We propose an algorithm that seems better equipped to handle such systems and prove upper and lower bounds on the update costs of graphs generated by it. Finally, we prove lower bounds on the update cost of any continuous group-key agreement scheme and multicast encryption scheme in a symbolic model.

4.1.1 The asymptotic setting

Given a set system $\mathcal{S} = (S_1, \ldots, S_k)$ over some base set $[n]$, we let $\mathcal{S}(N)$ denote the system with base set $[N]$ we get by considering each element in S with multiplicity N/n. E.g. if $\mathcal{S} = (\{1,2\},\{2,3\})$ then $\mathcal{S}(6) = (\{1,2,4,5\},\{2,3,5,6\})$.[2] Thus, as the

[2] $\mathcal{S}(N)$ is only well defined if N/n is an integer, we ignore this technicality as we will be interested in the case $N \to \infty$.

number of users N grows the relative sizes of the groups and their intersections remain fixed.

Let $s_i \coloneqq |S_i|/n$ denote the relative size of S_i and $s = \sum_{i=1}^{m} s_i$ be the average number of groups users are in. We assume wlog. that each user is in at least one group, implying $s \geq 1$. Let $\mathrm{Opt}(\mathcal{S})$ denote the update cost of the best key-graph for a set system $\mathcal{S}$ and $\mathrm{Triv}(\mathcal{S})$ the update cost of the Trivial algorithm (which makes a key-tree for every $S_i \in \mathcal{S}$). We will show that (the hidden constants in the big-Oh notation all depend on k, the number of groups).

$$\mathrm{Opt}(\mathcal{S}(N)) = N \log(N) + \Theta(N) \tag{4.1}$$

$$\mathrm{Triv}(\mathcal{S}(N)) = s \cdot N \log(N) - \Theta(N) \tag{4.2}$$

$$\text{thus } \frac{\mathrm{Triv}(\mathcal{S}(N))}{\mathrm{Opt}(\mathcal{S}(N))} = s - o(1) \tag{4.3}$$

As s is the average number of groups users are in, this shows that

> *asymptotically* (for a fixed set system $\mathcal{S}$ but with increasing number N of users) the update cost of an optimal key-derivation graph depends only on N (but not on $\mathcal{S}$). In this regime, the gain we get by using more cleverly chosen key-derivation graphs (as opposed to using a key-tree for every group) can be up to linear in s, the number of groups an average user is in, but not, say, the number of groups $|\mathcal{S}|$.

While we do not know how to efficiently find the best key graph for a given set system $\mathcal{S}$, in Section 4.4 we define a family $\mathcal{G}_{\mathrm{ao}}(\mathcal{S}(N))$ which is asymptotically optimal, i.e., matches Equation 4.1. Intuitively, it first partitions the universe of users $[N]$ into the sets of users that are members of exactly the same groups. More precisely, for $I \subseteq [k]$ let P_I be the set of users that are members of the groups specified by I. Then, the asymptotically optimal algorithm builds a balanced binary tree for every P_I, and in a second step connects the roots of these trees to the appropriate group keys by another layer of binary trees. For an illustration of the trivial and asymptotically optimal algorithms see Figure 4.1.

4.1.2 The non-asymptotic setting

Asymptotics can kick in slowly. The asymptotic setting gives a good idea about the efficiency we can expect once the number of users N is large compared to the number $k = |\mathcal{S}|$ of groups. Nevertheless, it should be noted that this asymptotic effect can kick in only slowly: assume the artificial example where for some small base set $[n]$ we have a set system $\mathcal{S} = \{S_1, \ldots, S_k\}$ with $k = 2^n - 1$ groups where for every

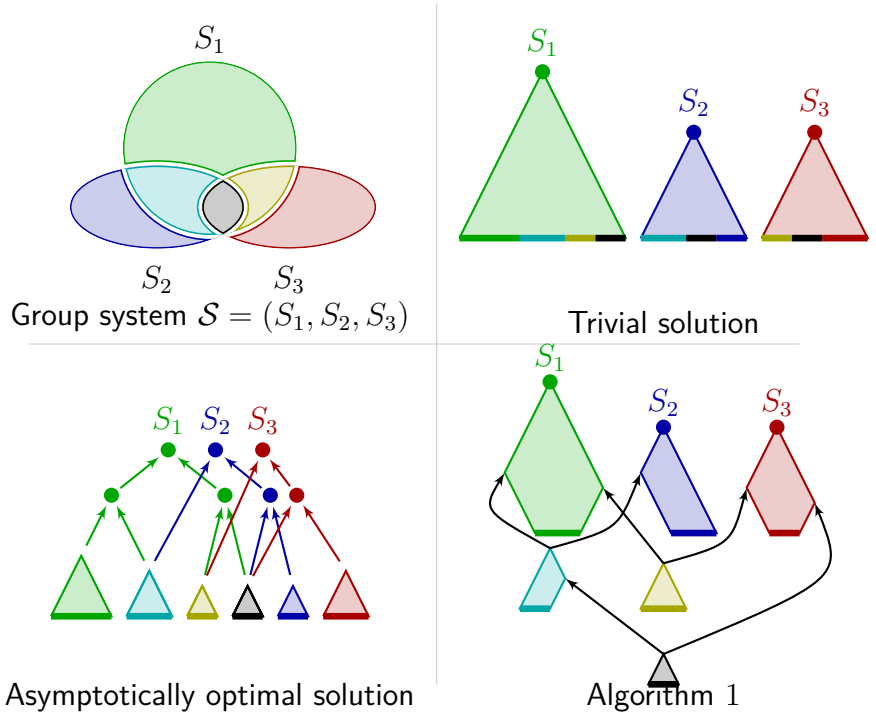

Figure 4.1: Key graphs for group systems. Top left; Venn diagram of the considered group system. Top right; trivial key graph using one balanced binary tree per group. Bottom left; Asymptotically optimal key graph using one balanced binary tree per partition P_I. Bottom right; asymptotically optimal key graph obtained using Algorithm 1. In the depictions of key trees the horizontal thick lines indicates the users' personal keys.

non-empty subset of users we have a group. Then each user is in 2^{n-1} groups and thus needs at least that many keys, and so the $\Theta(1)$ term in the asymptotic update cost $\log(N) + \Theta(1)$ of a single user is also at least 2^{n-1}. For the $\log(N)$ term to dominate we need $\log(N) \gg 2^{n-1}$, or $N \gg 2^{2^{n-1}}$, so the number of users needs to grow doubly exponential in the base set $[n]$.

Moving on to the non-asymptotic setting, consider a group system $\mathcal{S}$ for a fixed set of users $[N]$. The discussion above indicates that for $\mathcal{S}$ the asymptotic update cost per user of $\log(N)$ could be very far off the truth unless N becomes fairly large compared to the number of groups. This leaves the possibility that for concrete group systems where N is not huge relative to $\mathcal{S}$, already the trivial key-graph performs fairly well in practice. This, however, turns out to not be the case.

First, let us observe that the gap in update cost can never be larger than $\log(N)$, for

any $\mathcal{S}$ over $[N]$

$$\mathrm{Triv}(\mathcal{S}) \leq \log(N) \cdot \mathrm{Opt}(\mathcal{S}) \tag{4.4}$$

To see this we observe that the update cost for every user $i \in [N]$ is at most a factor $\log(N)$ larger in the trivial solution: a user i that is in $s_i = |\{S \in \mathcal{S} \ : \ i \in S\}|$ groups has an update cost of at least s_i in any key graph, in particular in $\mathrm{Opt}(\mathcal{S})$, and at most $\sum_{S \in \mathcal{S}, i \in S} \log(|S|) \leq s_i \cdot \log(N)$ in the trivial key graph.

In Section 4.4.2 we will show that this is not merely a theoretical gap by giving an example of a natural system $\mathcal{S}$ for which the update costs of both the trivial and the asymptotically optimal algorithms match the gap of $\log(N)$.

A greedy algorithm based on Huffman codes. The discussion above indicates that for set systems mapping groups that we might encounter in practice, one should not simply use an asymptotically optimal solution, but aim for a solution that is optimal, or at least close to optimal, for all instances.

Algorithm 1 that we propose in Section 4.5 is an algorithm for computing a key-graph given a set system $\mathcal{S}$. In a first step, the algorithm computes a "Boolean-lattice graph" for $\mathcal{S}$, and in a second iteratively runs the algorithm to compute Huffman Codes to compute the key graph. As the algorithm is basically a composition of greedy algorithms, it is very efficient. We leave it as an open question whether it really is optimal, and if not, whether there's an efficient (polynomial time) algorithm to compute $\mathrm{Opt}(\mathcal{S})$ and find the corresponding key graph for a given $\mathcal{S}$ in general.[3]

We present Algorithm 1 in Section 4.5 and discuss its connection to Boolean lattices. Then, we derive concrete lower and upper bounds on its update cost, that can serve as a good estimate on how much it saves compared to the trivial algorithm and the asymptotically optimal algorithm of Section 4.1.1. We further show that Algorithm 1 and a class of algorithms generalizing the approach taken are optimal in the asymptotic setting. While the same is true for the algorithm discussed in Section 4.1.1, Algorithm 1 seems better suited for practical applications as key-derivation graphs constructed by it reflect the hierarchical structure inherent to such systems. An example of a key graph generated by it is in Figure 4.1.

Our analysis concerns static group systems, but in Section 4.6 we show how known techniques that allow adding and removing users from groups in the settings of continuous group-key agreement and multicast encryption for a single group, can be adapted to key-derivation graphs generated by the greedy algorithm.

[3]The question whether a polynomial time algorithm for computing $\mathrm{Opt}(\mathcal{S})$ exists can be naturally asked in various ways. We discuss it in more detail in Section 4.10.

Lower bounds. To get a feeling for how close to optimal our approach is, we prove a lower bound on the average update cost for arbitrary schemes for continuous group-key agreement (in Section 4.7), and the related primitive of multicast encryption (in Section 4.9), that are based only on simple primitives such as encryption, pseudorandom generators, and secret sharing in a *symbolic security model*. This closely follows ideas from Micciancio and Panjwani [MP04], who considered such a symbolic model to analyze the worst-case update cost of multicast encryption schemes. We improve on their results by considering the setting of *multiple* potentially overlapping groups and proving a lower bound on the *average* communication complexity.

Our bound essentially shows that on average the cost of a user in any CGKA scheme or multicast encryption scheme for group system $S_1, \ldots, S_k$ constructed from the considered primitives satisfies

$$\mathrm{Upd}(\mathcal{G}) \geq \frac{1}{N} \cdot \sum_{\emptyset \neq I \subseteq [k]} |P_I| \cdot \log(|P_I|) \ ,$$

where $P_I \subseteq [N]$ is the set of users *exactly* in the groups specified by index set $I \subseteq [k]$. We consider it an interesting open question to either improve on this bound or to construct an algorithm matching it.

4.1.3 Related Work

On top of the related work discussed in Section 1.3, we discuss here some works that are of particular relevance for this chapter.

In the setting of a single group, key graphs have been used to construct secure multicast encryption, e.g. [WHA98, WGL00, CGI+99], and continuous group-key agreement (CGKA), e.g. ART [CCG+18] and TreeKEM[BBR+23] and all of it's variants. In the setting of multiple groups, the approach to use binary trees for every set of users that are members of exactly the same groups similarly to the asymptotically optimal algorithm, has been suggested in [MSAAA14, ZLC17]. However, the trees are then combined in a way that induces an overhead that is linear in the number of trees.

In [CHK21], Cremers et al. consider the post-compromise security guarantees of CGKA protocols for multiple groups. They show that in certain update scenarios, protocols based on pairwise channels have better healing properties than protocols based on key trees, as updates in a single group also benefit all subgroups of it. We stress that these issues do not arise in our approach, as updates in our setting are global and thus affect all groups the updating user is a member of.

The symbolic security model was first introduced by Dolev and Yao [DY83] and, as mentioned above, first used in this context by Micciancio and Panjwani [MP04] to

prove worst case bounds on the update cost of multicast encryption schemes using PRFs, secret sharing and symmetric encryption for a single group. In the context of CGKA schemes, it was used by Bienstock *et al.* [BDR20] to prove a lower bound on the communication cost of achieving PCS in 2 rounds for a single group, for protocols built using (dual) PRFs, (updatable) PKE and broadcast encryption. Recently, Auerbach *et al.* [ACNPPP23] extended the above results to the general setting of achieving PCS in arbitrary k rounds for protocols built from PRFs and PKE. Finally, Anastos *et al.* [AAB+24b] use it to prove a lower bound on the cost of adding and removing users in CGKA and ME schemes built from (dual) PRFs, secret sharing and encryption.

4.2 Preliminaries

4.2.1 Notation

We introduce some notation specific for this chapter.

Graph notation. Let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a directed acyclic graph (DAG). To node $v \in \mathcal{V}$ we associate the sets $\mathcal{A}(v) = \{v' \in \mathcal{V} \mid \exists \text{ path from } v' \text{ to } v\}$ of *ancestors* of v, and $\mathcal{D}(v) = \{v' \in \mathcal{V} \mid \exists \text{ path from } v \text{ to } v'\}$ of *descendants* of v. Here, we allow paths of length 0 and hence $v \in \mathcal{A}(v)$ and $v \in \mathcal{D}(v)$. Let $\mathcal{G}' = (\mathcal{V}', \mathcal{G}')$ be a subgraph of $\mathcal{G}$ and $v \in \mathcal{V}'$. We denote the set of parents of v by $\mathcal{P}(v)$. The set of *co-parents* $\mathcal{CP}(v, \mathcal{G}') \subseteq \mathcal{V}$ of v with respect to $\mathcal{G}'$ in $\mathcal{G}$ is the set of vertices that are parents of v in $\mathcal{G}$ but not in $\mathcal{G}'$. Even though there will be a notion of users, in this chapter we want to focus more on the properties of the key-derivation graphs and less on the real-world connection. Therefore, we will refer to them simply by integers $ß \in [N]$, as opposed to through their identifier ID, as we do in other chapters.

Probability distributions. Let X be a random variable with outcomes $x_1, \ldots, x_\ell$ with probability $p_1, \ldots, p_\ell$. Then we denote by $\mathbb{E}[X]$ its expectation and by $H(X) = -\sum_{i=1}^{\ell} p_i \log(p_i)$ its Shannon entropy.

4.2.2 Huffman Codes

Given a collection $v_1, \cdots, v_\ell$ of disconnected leaves of weight $w_1, \ldots, w_\ell \in \mathbb{N}$ a *Huffman Tree* is constructed as follows. From the set $\{v_1, \ldots, v_\ell\}$ two nodes v_{i_1}, v_{i_2} with the smallest weights are picked. Then a node v and edges $(v_{i_1}, v), (v_{i_2}, v)$ are added to the graph. v's weight is set to $w_{i_1} + w_{i_2}$ and the set of nodes to be considered updated to $\{v_1, \ldots, v_\ell\} \cup \{v\} \setminus \{v_{i_1}, v_{i_2}\}$. This step is repeated until all leaves are collected under a single root.

Since all nodes have indegree 2 the Huffman tree defines a prefix-free binary code for $(v_1, \ldots, v_\ell)$. We will make use of the following property of Huffman Codes.

Lemma 4.2.1 (Optimality of Huffman Codes [Huf52])**.** *Consider a Huffman tree $\mathcal{T}$ over leaves $v_1, \ldots, v_\ell$ of weight $w_1, \ldots, w_\ell \in \mathbb{N}$. Let $w = \sum_{i=1}^{\ell} w_i$ and let $U_{\mathcal{T}}$ denote the probability distribution that picks leaf v_i with probability w_i/w proportional to its weight. Then, if $\mathrm{len}(U_{\mathcal{T}})$ denotes the random variable measuring the length of the path from a leaf picked according to $U_{\mathcal{T}}$ to the root, we have that the average length of such paths is bounded by*

$$H(U_{\mathcal{T}}) \leq \mathbb{E}[\mathrm{len}(U_{\mathcal{T}})] \leq H(U_{\mathcal{T}}) + 1 \ .$$

4.3 Key-derivation Graphs for Multiple Groups

In this section we discuss key-derivation graphs for systems consisting of multiple groups. In Section 4.3.1 we briefly recall two applications of such graphs; continuous group-key agreement and multicast encryption. In Section 4.3.2 we formally define key-derivation graphs, discuss how key material in a graph is refreshed, and define its update cost.

4.3.1 Continuous Group-key Agreement and Multicast Encryption

Continuous group-key agreement. *Continuous group-key agreement* (CGKA) schemes [ACDT20] are an important building block in the construction of secure asynchronous group messaging schemes, as we have discussed previously. As the name indicates, the goal of a CGKA scheme is to establish a common key that is to be used to secure the communication between members of a group.

In this chapter, however, we are interested in the more general setting in which users $n \in [N]$ want to agree on keys for a system of groups $S_1, \ldots, S_k \subseteq 2^{[N]}$. After the groups have been established in a setup phase user n can use the procedure $\mathrm{Upd}(n)$ to produce an update message that rotates the key material known to them, thus eliminating any keys that may have leaked during a compromise. As before, this update message is broadcast to the other users using the untrusted delivery server. Given their own secret keys, users are then able to retrieve the refreshed keys that should be known to them. A natural goal to aim for is to minimize the communication cost incurred by such update messages.

Naturally, one would like to additionally support dynamic operations also in this setting. While here we will focus on the update costs of schemes for a system of static groups, in Section 4.6 we show that the known techniques of blanking and unmerged leaves

used in the MLS protocol [BBR+23] can be adapted to schemes obtained from our approach.

Multicast encryption. The goal of a *multicast encryption* scheme [WHA98, WGL00, CGI+99] is to establish a key for a group of users to enable them to decrypt ciphertexts broadcast to the group. Every user holds a personal long-term key, but opposed to CGKA there also exists a central authority that has access to all secret key material. After a setup phase, the central authority is able to add and remove users from the group by refreshing key material and broadcasting messages to the group. The central goal in the construction of multicast schemes is to minimize the communication complexity incurred by such operations. Typically, multicast encryption schemes also rely on key-derivation graphs.

As in the case of CGKA, we are interested in the more general setting of a system of potentially overlapping groups of users.

4.3.2 Key-derivation Graphs

We now discuss key-derivation graphs, which can be seen as a generalization of ratchet trees. In our exposition we will focus on graphs for continuous group-key agreement. At the end of the section we discuss the differences to graphs for multicast encryption.

Consider a set of parties $[N]$ and a collection $\mathcal{S} \subseteq 2^{[N]}$ of subgroups of $[N]$. A key-derivation graph (kdg) for $[N]$ and $\mathcal{S}$ organizes key pairs in a way that allows the members of a particular subgroup to agree on a key, and further enables parties to refresh the key material known to them. Every node v in the graph is associated to a key pair $(\mathrm{pk}_v, \mathrm{sk}_v)$ of a public-key encryption scheme $(\mathrm{PKE.Gen}, \mathrm{PKE.Enc}, \mathrm{PKE.Dec})$, and edges (v, v') indicate that parties with access to sk_v also posses $\mathrm{sk}_{v'}$. The personal keys of users correspond to sources and every group is represented by a node that holds the corresponding secret group key. We formalize the structural requirements on the graph in the multi-group setting as follows.

Defintion 4.3.1. *Let $N \in \mathbb{N}$, $\mathcal{S} \subseteq 2^{[N]}$, and $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ a DAG. We say that $\mathcal{G}$ is a* key-derivation graph *for universe of elements $[N]$ and groups $\mathcal{S}$ if*

1. *For every $n \in [N]$ there exists a source $v_n \in \mathcal{V}$ and for every $S \in \mathcal{S}$ there exists a node $v_S \in \mathcal{V}$. We further require that $v_n \neq v'_n$ for $n \neq n'$.*
2. *For $n \in [N]$ and $S \in \mathcal{S}$ we have $v_S \in \mathcal{D}(v_n)$ exactly if $n \in S$.*

In the definition above node v_n correspond to user n's personal key, and nodes v_S to group keys. The second property encodes correctness and security, intuitively saying that n is able to derive the group key of S exactly if $n \in S$.

Updates. Let $\mathcal{G}$ be a key-derivation graph for $[N]$ and $\mathcal{S}$. If party n wants to perform an update they have to refresh all key-material corresponding the subgraph $\mathcal{D}(v_n)$ known to them and communicate the change to the other parties. To this end they pick a spanning tree $\mathcal{T}_n = (\mathcal{V}', \mathcal{E}')$ of $\mathcal{D}(v_n)$, as well as a random seed Δ_{v_n}. Then starting from the source v_n, if v' is the ith child of node v they define the seed of v' as $\Delta_{v'} = \mathrm{H}(\Delta_v, i)$, where H is a hash function. $\Delta_{v'}$ is then used to derive a new key-pair $(\mathrm{pk}_{v'}, \mathrm{sk}_{v'}) \leftarrow \mathrm{PKE.Gen}(\Delta_{v'})$ for v'. Finally, for every $v \in \mathcal{V}'$ and every co-parent $v' \in \mathcal{CP}(v, \mathcal{T}_n)$, n computes the ciphertext $c_{v,v'} = \mathrm{PKE.Enc}(\mathrm{pk}_{v'}, \Delta_v)$. The set of all ciphertexts together with the set of new public keys forms the update message. Finally, n deletes all seeds Δ_v.

We now show that the construction preserves correctness, i.e., users $n' \neq n$ are able to deduce all new secret keys in $\mathcal{D}(v_{n'})$ from the update message and thus in particular the group keys of all groups they are a member of. To this end, let $v \in \mathcal{D}(v_n) \cap \mathcal{D}(v_{n'})$. Then there exists a path $(v_{n'} = v_1, \ldots, v_\ell = v)$ in $\mathcal{D}(v_{n'})$. Let i be maximal with $v_i \notin \mathcal{D}(v_n)$ (Note that such i must exist as $v_{n'}$ is a source). By maximality of i the node v_i must be a coparent of v_{i+1} with respect to $\mathcal{D}(v_n)$. Thus, the update message contains an encryption of Δ_{v_i+1} to pk_{v_i}. As sk_{v_i} was not replaced by the update and is known to n' the user can recover $\Delta_{v_{i+1}}$ and in turn $\mathrm{sk}_{v_{i+1}}$. Now, n' can recover the remaining $\Delta_{v_{i+2}}, \ldots, \Delta_{v_\ell}$ and the corresponding secret keys as the seeds were either derived by hashing or, in the case that v_{j+1} is a coparent of v_j with respect to $\mathcal{D}(v_n)$, encrypted to the new key pk_{v_j}, the secret key of which was already recovered by n'.

Update cost. Using the size of ciphertexts as a unit, the update cost of n is given by $\mathrm{Upd}(n) = \sum_{v \in \mathcal{T}_n} |\mathcal{CP}(v, \mathcal{T}_n)| = \sum_{v \in \mathcal{T}_n} (|\mathcal{P}(v)| - 1)$. Note that this quantity is independent of the particular choice of spanning tree $\mathcal{T}_n$. In this work we are interested in minimizing the *average* update cost, assuming that every user updates with the same probability. We define the *total update cost* $\mathrm{Upd}(\mathcal{G}) = \sum_{n \in [N]} \mathrm{Upd}(n)$ of $\mathcal{G}$. Note that $\mathrm{Upd}(\mathcal{G})/N$ is the average update cost of a user, and we can thus focus on trying to minimize $\mathrm{Upd}(\mathcal{G})$, which will allow for easier exposition. The following lemma shows that we can restrict our view to graphs in which every non-source has indegree 2. Note, that for graphs $\mathcal{G}$ with this property we have $|\mathcal{CP}(v, \mathcal{T}_n)| = 1$ for every n, $\mathcal{T}_n$, and $v \in \mathcal{T}_n$ that is not a source and thus in this case we can compute the update cost as

$$\mathrm{Upd}(\mathcal{G}) = \sum_{n \in [N]} (|\mathcal{T}_n| - 1) = \sum_{n \in [N]} (|\mathcal{D}(n)| - 1) = \sum_{n \in [N]} |\mathcal{D}(n)| - N \ . \tag{4.5}$$

Lemma 4.3.2. *Let $n \in \mathbb{N}$, $\mathcal{S} \subseteq 2^{[N]}$, and $\mathcal{G}$ a key-derivation graph for $[N]$ and $\mathcal{S}$. Then there exists a key-derivation graph $\mathcal{G}'$ for $[N]$ and $\mathcal{S}$ satisfying $\mathrm{Upd}(\mathcal{G}') \leq \mathrm{Upd}(\mathcal{G})$ such that for every non-source $v \in \mathcal{V}'$ we have $\mathrm{indeg}(v) = 2$.*

Proof. We first show that we can iteratively decrease the indegree of nodes in $\mathcal{G}$ in a way that preserves correctness and only improves the update cost. Thus let $v \in \mathcal{V}$

and $\mathcal{P}(v) = \{v_1, \ldots, v_k\}$ with $k \geq 3$ and consider the graph $\mathcal{G}'$ with $\mathcal{V}' = \mathcal{V} \cup \{v'\}$ and $\mathcal{E}' = \mathcal{E} \cup \{(v_1, v',), (v_2, v'), (v', v)\} \setminus \{(v_1, v), (v_2, v)\}$. Note that the correctness requirements of Definition 4.3.1 are unaffected by this modification. Let $n \in [N]$. If $v \notin \mathcal{D}(n)$ n's update cost remains the same after the change. Thus, assume $v \in \mathcal{D}(v)$. In $\mathcal{G}$ we have $|\mathcal{P}(v)| = k$. In $\mathcal{G}'$, on the other hand, $|\mathcal{P}(v)| = k - 1$ and $|\mathcal{P}(v') = 2|$. Thus if n's spanning trees in $\mathcal{G}$ uses edge (v_1, v) or (v_2, v), then their update cost remains unchanged in $\mathcal{G}'$. If, on the other hand, neither (v_1, v) nor (v_2, v) lie in n's spanning trees, then their update cost decreases by one. after repeating the step sufficiently many times we end up with a graph $\mathcal{G}'$ with $\mathrm{indeg}(v) \leq 2$ for all $v \in \mathcal{V}'$.

Finally note that non-sources with indegree 1 can be simply merged with their parent, which does not affect correctness and update cost of the graph. □

Key-derivation graphs for multicast encryption. Opposed to kdgs for CGKA key-derivation graphs for multicast encryption rely on symmetric encryption. Let (E, D) be a symmetric encryption scheme. Every node v in a kdg $\mathcal{G}$ for $[N]$ and $\mathcal{S}$ is associated to a key ssk_v[4], and an edge (v, v') indicates that a party with access to ssk_v knows $\mathrm{ssk}_{v'}$. We require structural requirements on $\mathcal{G}$ that are analogous to Definition 4.3.1. Updates with respect to leaf v_n, which for multicast encryption are computed by the central authority, and their update cost, are defined analogous to the setting of CGKA as well.

While the main goal of multicast encryption is not to recover from compromise of keys by updating, but instead to allow the central authority to dynamically change the structure of the groups $S_1, \ldots, S_k$, the notion of an update with respect to a leaf v_n still turns out to be useful. Assume that the central authority performed an update for v_n starting with seed Δ. We can distinguish two cases. If Δ is not known to the owner n of leaf v_n then n lost access to all keys corresponding to $\mathcal{D}(v_n)$. Thus, by updating, the central authority can remove a party from *all* groups they are a member of. Assume on the other hand that the leaf was previously unpopulated and that Δ can be derived from n's long term key. Then n gained access to all group keys that can be reached from v_n. In Section 4.6 we discuss how updates can be used as the basic building block of implementing more fine grained operations, i.e., adding or removing a user from particular group S_i. The efficiency of these operations is significantly determined by the update cost as defined in this section.

[4]Outside of this chapter, this notation is used to denote the secret key of a signature scheme. Nevetheless, since signature keys are not used in this chapter, and symmetric keys rarely mentioned outside of it, we consider it acceptable to reuse the notation.

4.3.3 Security

The main focus of this work is to investigate the communication complexity of key-derivation graphs for group systems. We do not give formal security proofs in this work. The structural requirements on kdgs and definition of update procedures are chosen with the goal of the resulting CGKA to achieve *post-compromise forward-secrecy* (PCFS) [ACJM20] roughly corresponding to *post-compromise security* (PCS) and *forward-secrecy* (FS) simultaneously. In the following paragraphs we provide an intuition on the security properties of kdgs. For ease of exposition we will discuss PCS and FS separately instead of PCFS.

Recall that CGKA schemes constructed from kdgs employ further mechanisms to ensure authenticity and prevent a malicious sever to send users inconsistent update messages. We consider the construction of such mechanisms as well as a formal security analysis of kdgs to be important open questions for future work.

Preserving the graph invariant. We first discuss how updates preserve the invariant, that users n know exactly the secret keys corresponding to $\mathcal{D}(v_n)$, which by Condition 2 of Definition 4.3.1 implies that n will never be able to derive a group key for some group they are not a member of. Note that if n is the updating user then they will only replace keys in $\mathcal{D}(v_n)$. If n receives an update message, on the other hand, then they will only be able to recover a key sk_v if either the corresponding seed Δ_v was encrypted to a key known to n or if Δ_v was derived by hashing from a seed $\Delta_{v'}$ recoverable by n. By iteratively applying this argument to $\Delta_{v'}$ we obtain that there must exist some $\Delta_{v''}$ that was encrypted to a key known to n such that v'' has a path to v. Thus, it must hold that $v \in \mathcal{D}(v_n)$. (Note that the one-wayness of the used hash function ensures that seeds derived by hashing can only be recovered from each other in the correct direction.)

Post-compromise security. The goal of PCS is to allow users whose secret state has been exposed to recover from this exposure by performing an update. Using the example of a single compromised user we now discuss how kdgs for group systems achieve this property. Assume that an adversary knows exactly the secret state of user n, i.e., all keys sk_v for $v \in \mathcal{D}(v_n)$, and that n then performs an update. Then the adversary is not able to deduce any of the replaced keys: Note that the initial random seed Δ_{v_n} is not encrypted to any key and thus cannot be leaked to the adversary. Thus, all other seeds Δ_v can only be derived by the adversary if Δ_v itself, or a seed from which Δ_v was derived by iterated hashing was encrypted to a key known to the adversary. However, the adversary only knows the keys corresponding to $\mathcal{D}(v_n)$ before the update, and those keys were replaced by freshly sampled ones before computing the

ciphertexts. Thus, seeds are encrypted to either "old" keys not known to the adversary or new keys, and so after the update all keys are secure again.

Forward secrecy. Forward secrecy requires that compromising a user's secret state does not allow the adversary to recover previous group keys. In key-derivation graphs old keys get deleted over time providing a limited form of forward-secrecy. Concretely, if a user n is corrupted all group keys before their last update remain secure. This holds, since seeds that were generated before this point in time and can be used to recover group keys were encrypted to keys no longer in n's memory. Note however, that group keys generated in between n's last update and the time of n's corruption might leak to the adversary. For example, a seed from which such keys can be derived might have been encrypted to the key sk_{v_n}, which remained unchanged until the corruption.

Improved forward secrecy using supergroups. Recall that CGKA constructions employing kdgs like TreeKEM [BBR18] rely on an additional mechanism to improve their forward-secrecy guarantees. Namely, to use application secrets, resulting from hashing the epoch's shared secret into a hash chain of application secrets, to encrypt messages. This allows users to gain the advantage of forward secrecy not only by issuing but also by processing updates of other users in S

In the setting of a group system $\mathcal{S}$ we can further improve on this: Consider some group $S \in \mathcal{S}$ and let $S_1, \ldots, S_\ell$ be the maximal (with respect to inclusion) groups in $\mathcal{S}$ that contain S. We denote the application secrets for S and the S_i by K_S and K_{S_i} respectively. Now, whenever a member of any of the S_i issues an update the application secret of S is updated to $K_S \leftarrow \mathrm{H}_2(\mathrm{sk}_{v_S}, K_{S_1}, \ldots, K_{S_\ell})$.[5] Note that for every i since $S \subseteq S_i$ all members of S do indeed have access to K_{S_i} and thus are able to compute K_S, and that an update by users in S implies that all S_i are updated as well. The effect of this modification is that even updates by users *outside* of S —more precisely in any of the sets $S_i \setminus S$— imply forward secrecy of users in S. Note that this is in particular helpful in the case where $|S| \ll |S_i|$ and updates in the large group occur much more frequently than in the small group, for example in the case of two members of a large group having a private conversation.

4.3.4 The Trivial Algorithm

To construct a key-derivation graph for a single group S the parties $n \in S$ are typically arranged as the leaves of a balanced binary tree $\mathcal{T}$. The tree's root serves as the group key. In this case the length of paths from leaf to root is at most $\lceil \log(|S|) \rceil$ and in turn $\mathrm{Upd}(\mathcal{T}) \leq |S| \cdot \lceil \log(|S|) \rceil$. On the other hand, $\mathcal{T}$ defines a prefix-free binary code for

[5] Regarding PCFS it might even be advantageous to include $K_{S'}$ for *all* $S' \supseteq S$.

the set S. Thus, by Shannon's source coding theorem the average length of paths from leaf to root is at least $\log(|S|)$ which implies $\mathrm{Upd}(\mathcal{T}) \geq |S| \cdot \log(|S|)$.

An algorithm for multiple groups. A trivial approach to construct a key derivation graph for parties $[N]$ and group system $\mathcal{S} = \{S_1, \ldots, S_k\}$ is to simply apply the method described above to all S_i in parallel. That is, for $i \in [k]$ construct a balanced binary tree $\mathcal{T}_i$ with $|S_i|$ leaves such that for $n \in [N]$ the node v_n is a leaf of exactly the trees $\mathcal{T}_i$ with $n \in S_i$. Let $\mathcal{G}$ denote the resulting graph. The conditions of Definition 4.3.1 clearly hold and we can bound the total update cost of $\mathcal{G}$ by

$$\sum_{i \in [k]} |S_i| \cdot \log(|S_i|) \leq \mathrm{Upd}(\mathcal{G}) \leq \sum_{i \in [k]} |S_i| \cdot \lceil \log(|S_i|) \rceil \ .$$

Further, the update cost of a single user $n \in [N]$ is bounded by $\mathrm{Upd}(n) \leq \sum_{i : n \in S_i} \lceil \log(|S_i|) \rceil$.

4.4 Key-derivation Graphs in the Asymptotic Setting

In this section we investigate the update cost of key-derivation graphs for multiple groups in an asymptotic setting. More precisely, for a system consisting of a fixed number of groups, we consider the setting in which the number of users tends to infinity while the relative size of the groups stays constant. In Section 4.4.1 we first compute the asymptotically optimal update cost of key-derivation graphs and then show that the trivial algorithm does not achieve it. We then present an algorithm achieving the optimal update cost. In Section 4.4.2 we show that both approaches can perform badly for *concrete* group systems.

4.4.1 Key-derivation Graphs in the Asymptotic Setting

We investigate the update cost of key derivation graphs in an asymptotic setting. That is, we consider N parties that form a subgroup system $\mathcal{S} = \{S_1, \ldots, S_k\}$ and fix values $p_I \in [0, 1]$ for $I \subseteq [k]$ that indicate the fraction of users that are members of exactly the groups specified by I.

More precisely, let $k \in \mathbb{N}_{\geq 2}$ be fixed and let $\{p_I\}_{I \subseteq [k]}$ be such that $\sum_{I \subseteq [k]} p_I = 1$. For $N \in \mathbb{N}$ let $\mathcal{S}(N) = \{S_1(N), \ldots, S_k(N)\} \subseteq 2^{[N]}$ be a subgroup system that satisfies $|P_I(N)| = N \cdot p_I$ for all I, where $P_I(N) = \bigcap_{i \in I} S_i(N) \setminus \bigcup_{j \in [k] \setminus I} S_j(N)$ is the set of users exactly in the groups specified by I.[6] Throughout this section we assume that

[6] $\mathcal{S}(N)$ is only well defined if $N \cdot p_I$ is an integer for all I, we ignore this technicality as we are interested in the case $N \to \infty$.

$p_\emptyset = 0$, i.e., every user is in at least one group, and that at least two groups are non-empty. We are interested in the update cost of key-derivation graphs for $\mathcal{S}(N)$ when N tends to infinity.

Lower bound in the asymptotic setting. We first compute a lower bound on the update cost of kdgs in the asymptotic setting. The bound follows from the following combinatorial result on *concrete* graphs, that will also turn out to be useful for our symbolic lower lower bound of Section 4.7. Recall that for graphs $\mathcal{G}' \subseteq \mathcal{G}$ and a vertex v the set $\mathcal{CP}(v, \mathcal{G}')$ is the set of co-parents of v with respect to $\mathcal{G}'$ in $\mathcal{G}$.

Lemma 4.4.1. *Let $M \in \mathbb{N}$ be fixed, $S_1, \ldots, S_k \subseteq [M]$, and let $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ be a DAG such that there exist pairwise disjoint sets of sources V_n, $n \in [M]$, and nodes v_{S_i}, $i \in \{1, \ldots, k\}$ such that*

$$n \in S_i \quad \Rightarrow \quad \exists v_n \in V_n \text{ such that there is a path from } v_n \text{ to } v_{S_i} \ .$$

Further let $\mathcal{T}_n$ be a spanning forest of $\mathcal{D}(V_n) = \bigcup_{v_n \in V_n} \mathcal{D}(v_n)$. Then

$$M \cdot \mathbb{E}\Big[\sum_{v \in \mathcal{T}_n} |\mathcal{CP}(v, \mathcal{T}_n)| \Big] \geq \sum_{\emptyset \neq I \subseteq [k]} |P_I| \cdot \log(|P_I|) \ ,$$

where the expectation is to be understood with respect to the uniform distribution on $[N]$.

Proof. As a first step we show that we may assume that all V_n consist of a single source v_n. Indeed, we could replace $\mathcal{G}$ with a graph $\mathcal{G}'$ that for every n has an additional source v_n which has outgoing edges to all elements of V_n. As now all former sources have indegree 1 and all other nodes are unaffected $\sum_{v \in \mathcal{T}_n} |\mathcal{CP}(v, \mathcal{T}_n)|$ is the same in both graphs, and every bound on for $\mathcal{G}'$ carries over to $\mathcal{G}$.

Further, using the same argument as in the proof of Lemma $4.3.2$ we can replace $\mathcal{G}'$ by a graph $\mathcal{G}''$ satisfying the same correctness properties as $\mathcal{G}'$ such that all non-sources v of $\mathcal{G}''$ satisfy $\mathrm{indeg}(v) = 2$ and $\sum_{v \in \mathcal{T}_n} |\mathcal{CP}(v, \mathcal{T}_n)|$ can only decrease, implying that every bound for $\mathcal{G}''$ carries over to $\mathcal{G}'$ and in turn to $\mathcal{G}$. Thus assume that $\mathcal{G}$ satisfies $V_n = \{v_n\}$ for all n and that all non-sources have indegree 2.

Note that

$$\begin{aligned} M \cdot \mathbb{E}[\sum_{v \in \mathcal{T}_n} |\mathcal{CP}(v, \mathcal{T}_n)|] &= \sum_{n \in [N]} \sum_{v \in \mathcal{T}_n} |\mathcal{CP}(v, \mathcal{T}_n)| \\ &= \sum_{\emptyset \neq I \subseteq [k]} \sum_{n \in P_I} \sum_{v \in \mathcal{T}_n} |\mathcal{CP}(v, \mathcal{T}_n)| \ , \end{aligned}$$

where we used that the P_I form a partition of $[M]$ and that $P_\emptyset = \emptyset$. Thus, to prove the lemma it suffices to show that for every nonempty $I \subseteq [k]$ we have $\sum_{n\in P_I}\sum_{v\in\mathcal{T}_n}|\mathcal{CP}(v,\mathcal{T}_n)| \geq |P_I|\cdot\log(|P_I|)$. Fix a nonempty index set $I \subseteq [k]$ and let $i \in I$. By assumption for all $n \in P_I$ there exists a path $\mathcal{P}$ from v_n to v_{S_i}. Thus there exits a subgraph $\mathcal{G}'$ of $\mathcal{G}$ in which every v_n has exactly one path to v_{S_i}.

As all v_n are sources and $\mathrm{indeg}(v) \leq 2$ for all $v \in \mathcal{G}'$ the graph $\mathcal{G}'$ defines a prefix-free binary code for the set P_I. By Shannon's source coding theorem this implies that the average length of the paths from source to sink in $\mathcal{G}'$ is at least $H(U_{P_I}) = \log(|P_I|)$, where U_{P_I} denotes the uniform distribution over P_I. Summing over all elements of P_I we obtain

$$\sum_{n\in P_I}\sum_{v\in\mathcal{T}_n}|\mathcal{CP}(v,\mathcal{T}_n)| = \sum_{n\in P_I}(|\mathcal{T}_n|-1) \geq |P_I|\cdot\log(|P|_I) \ ,$$

where in the equality we used that all non sources have indegree 2 and in the inequality that $\mathcal{T}_n$ contains paths of average length $\log(P_I)$.

□

Note that Lemma 4.4.1 in the case $|V_n| = 1$ for all n can be seen as a lower bound on the total update cost of key-derivation graphs as defined in Section 4.3 since $M\cdot\mathbb{E}[\sum_{v\in\mathcal{T}_n}|\mathcal{CP}(v,\mathcal{T}_n)|] = \sum_{v\in\mathcal{T}_n}|\mathcal{CP}(v,\mathcal{T}_n)|$.

Turning to the asymptotic setting we have

$$\begin{aligned}\sum_{I\subseteq[k]} N\cdot p_I\cdot\log(N\cdot p_I) &= N\cdot\sum_{I\subseteq[k]} p_I\log(N) + N\cdot\sum_{I\subseteq[k]} p_I\log(p_I)\\ &= N\log(N) + N\cdot\sum_{I\subseteq[k]}\log(p_I) = N\log(N)+\Theta(N) \ ,\end{aligned}$$

where we used that $\sum_I p_I = 1$. As we will show below, there exist key-derivation graphs matching this bound. We conclude that the optimal update cost in the asymptotic setting only depends on the overall number of users but not the particular set system:

$$\mathrm{Opt}(\mathcal{S}(N)) = N\log(N)+\Theta(N) \ .$$

Note, however, that the term $\Theta(N)$ hides a constant (with respect to N), that can be *exponential in* k.

Asymptotic update cost of the trivial algorithm. The trivial algorithm constructs a separate balanced binary tree for every group $S_i(N)$. For $i \in [k]$ let s_i be such that $N\cdot s_i = |S_i(N)|$ and further let $s = \sum_{i=1}^k s_i$ be the average number of groups a user are member of. Then, we can bound the update cost $\mathrm{Triv}(\mathcal{S}(N))$ of the trivial algorithm in the asymptotic setting as follows, showing that is does not match the optimal cost.

Claim 1. *For $I \subseteq [k]$ let $p_I \in [0,1]$ be such that $\sum_{I \subseteq [k]} p_I = 1$ and $p_\emptyset = 0$. Let $\mathcal{S}(N)$ be the corresponding group system and s_i, s as defined above. Then*

$$\mathrm{Triv}(\mathcal{S}(N)) = s \cdot N \log(N) + \Theta(N) \ .$$

Proof. As discussed in Section 4.3.4, key-derivation graphs $\mathcal{G}$ for $\mathcal{S}(N)$ constructed by the trivial algorithm satisfy

$$\sum_{i \in [k]} |S_i(N)| \cdot \log(|S_i(N)|) \leq \mathrm{Upd}(\mathcal{G}) \leq \sum_{i \in [k]} |S_i(N)| \cdot \lceil \log(|S_i(N)|) \rceil \ .$$

Thus, on the one hand,

$$\begin{aligned}\mathrm{Triv}(\mathcal{S}(N)) &\geq \sum_{i=1}^{m} (s_i \cdot N) \cdot \log(N \cdot s_i) = \sum_{i=1}^{m} (s_i \cdot N) \cdot \log(N) + \log(s_i) \\ &= \sum_{i=1}^{m} s_i \cdot N \log N + \sum_{i=1}^{m} s_i \cdot N \cdot \log(s_i) = s \cdot N \log(N) + \Theta(N)\end{aligned}$$

and, on the other hand,

$$\begin{aligned}&\mathrm{Triv}(\mathcal{S}(N)) \\ \leq &\sum_{i=1}^{m} (s_i \cdot N) \cdot \log(N \cdot s_i) + O(N) = \sum_{i=1}^{m} (s_i \cdot N) \cdot \log(N) + \log(s_i) + O(N) \\ = &\sum_{i=1}^{m} s_i \cdot N \log N + \sum_{i=1}^{m} s_i \cdot N \cdot \log(s_i) + O(N) \leq s \cdot N \log(N) + O(N) \ .\end{aligned}$$

□

An asymptotically optimal graph. We will sketch how to construct an asymptotically optimal key graph $\mathcal{G}_{\mathrm{ao}}(N)$ for a given set system $\mathcal{S}$ over $[n]$. In a first step, for every I with $P_I(N) \neq \emptyset$, the algorithm constructs a balanced binary tree with root v_I using as leafs the elements of $P_I(N)$. Then, in a second step, for every group $S_i(N)$ it builds a balanced binary tree with root v_{S_i} using as leafs the nodes $\{v_I \mid I : i \in I\}$. An illustration of the algorithm's working principle is in Figure 4.1. Correctness of the construction follows by inspection.

To bound the update cost $\mathrm{Upd}(\mathcal{G}_{\mathrm{oa}}(N))$ we split it in two parts; the first accounts for the contribution of the nodes generated during the first step, the second for the contribution of the second step. As $\sum_I p_I = 1$, the first part contributes at most $\sum_{I \subseteq [k]} p_I \cdot N \cdot \log(N \cdot p_I) \leq N \cdot \log N$, while the contribution of the second part for every single user is constant as $\{v_I\}$ is independent of N, implying that with respect to the total update cost it is $\Theta(N)$. Thus, overall we get $\mathrm{Upd}(\mathcal{G}_{\mathrm{oa}}(N)) \leq N \cdot \log N + \Theta(N)$, matching the optimal update cost.

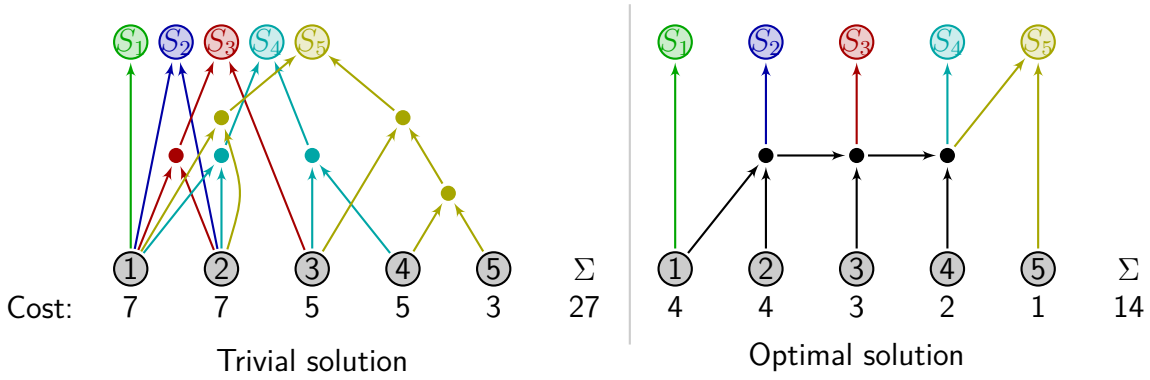

Figure 4.2: Illustration of $\mathrm{Triv}(\mathcal{S}_N^{\uparrow})$ (left) and $\mathrm{Opt}(\mathcal{S}_N^{\uparrow})$ for $N = 5$. For each user, the update cost (i.e., the indegree 2 nodes reachable) is indicated.

4.4.2 Update Cost for Concrete Group Systems

Now consider a concrete group system $\mathcal{S} = \{S_1, \ldots, S_k\}$ for a fixed set of users $[N]$. As already discussed in Section 4.1.2, it is possible that the number k of groups can be as large as $2^N - 1$. Thus, for concrete group systems the asymptotic update cost per user of $\log(N)$ (that contains hidden constants dependent on k) derived in Section 4.4.1 could be very far off the truth unless N becomes fairly large compared to the number of groups. This leaves the possibility that in the case where N is not huge relative to k, already the trivial key-graph performs fairly well in practice. In this section we show that this is not the case by giving an example where not only the trivial key-graph (which has a balanced tree for every set), but also our asymptotically optimal $\mathcal{G}_{\mathrm{oa}}$, perform poorly.

Recall that by Equation 4.4 the update costs of the trivial and optimal solutions always satisfy $\mathrm{Triv}(\mathcal{S}) \leq \log(N) \cdot \mathrm{Opt}(\mathcal{S})$. The above argument seems very loose, but we show an example where we indeed have a gap of $\approx \log(N) - 1$ and thus almost match this seemingly loose $\log(N)$ bound. Define the "hierarchical" set system $\mathcal{S}_N^{\uparrow}$ over $[N]$ as

$$\mathcal{S}_N^{\uparrow} \coloneqq \{S_1, \ldots, S_N\} \quad \text{where} \quad S_i = \{i, i+1, \ldots, N\} \ .$$

Note that while $\mathcal{S}_N^{\uparrow}$ is defined for all N, it is not asymptotic in the sense discussed in Section 4.4.1, as the number of groups grows with the number of users N. Further, for this group system the key derivation graphs output by the trivial and asymptotically optimal algorithms coincide, as for every P_I with $P_I \neq \emptyset$ we have $|P_I| = 1$. As the optimal solution for $\mathcal{S}$ is just a path, as illustrated in Figure 4.2, we obtain update

costs of

$$\mathrm{Triv}(\mathcal{S}_N^{\uparrow}) = \sum_{i=1}^{N} i\log(i) \approx \frac{N^2}{2}\log(N) \quad \text{and} \quad \mathrm{Opt}(\mathcal{S}_N^{\uparrow}) = \sum_{i=1}^{N} i = \frac{N(N+1)}{2} \approx \frac{N^2}{2} \ .$$

Thus $\mathrm{Triv}(\mathcal{S}_N^{\uparrow})/\mathrm{Opt}(\mathcal{S}_N^{\uparrow}) \approx \log(N)$ matching the (4.4) bound. An interesting observation is the fact that an optimal solution can have much larger *depth* than the trivial one: for $\mathcal{S}_N^{\uparrow}$ the depth of the optimal solution is N, while in the trivial solution it is just $\log(N)$. The discussion above indicates that neither the trivial nor the asymptotically optimal algorithm are well-equipped to handle certain group systems. In the following section we propose an algorithm that is not only asymptotically optimal, but also generates key-derivation graphs better reflecting the hierarchical nature of group systems, and, in particular, recovers the optimal solution for the example above.

4.5 A Greedy Algorithm Based on Huffman Codes

In this section we propose an algorithm to compute key-derivation graphs for group systems. Its formal description is in Section 4.5.1. In Section 4.5.2 we compute bounds on its total update cost and compare it to the trivial algorithm and the asymptotically optimal algorithm of Section 4.4.1 and in Section 4.5.3 we compute bounds on its worst-case update cost. Finally, in Section 4.5.4 we show that the algorithm, as well as a class generalizing it, are asymptotically optimal.

4.5.1 Algorithm Description

We now describe Algorithm 1 that, on input parties $[N]$ and a set of groups $\mathcal{S} \subseteq 2^{[N]}$, constructs a key-derivation graph. Its formal description is in Figure 4.3.

Conceptually, the algorithm proceeds in two phases. The first phase (lines 1 to 11) determines the macro structure of the key-derivation graph. For reasons explained below we will refer to the graph generated in this phase as the *lattice graph*. In the second phase (lines 12 to 20), sources for the individual users are added at the correct position in the lattice graph, which afterwards is binarized to reduce the update size.

More precisely, at the beginning of the first phase the algorithm initializes a graph $\mathcal{G} = (\mathcal{V}, \mathcal{E})$ consisting of isolated nodes $v_{S'}$ with $S' \in \mathcal{S}$ that, looking ahead, will represent the group keys. Every node $v_{S'}$ is associated to a set $\mathrm{S}(v_{S'})$ that is initialized to group S'. The algorithm then determines nodes v_1, v_2 such that the intersection of their associated sets is maximal and adds a node v_3 as well as the edges $(v_3, v_1), (v_3, v_2)$ to the graph. The associated set of v_3 is set to $\mathrm{S}(v_1) \cap \mathrm{S}(v_2)$ and the associated sets of v_1 and v_2 are updated to $\mathrm{S}(v_1) \setminus \mathrm{S}(v_3)$ and $\mathrm{S}(v_2) \setminus \mathrm{S}(v_3)$ respectively. This step is repeated until the associated sets of all nodes are pairwise disjoint.

Input: $(N, \mathcal{S})$

$\mathcal{G} = (\mathcal{V}, \mathcal{E}) \leftarrow (\emptyset, \emptyset)$
for $S' \in \mathcal{S}$
  $\mathcal{V} \leftarrow \mathcal{V} \cup \{v_{S'}\}$
  $\mathrm{S}(v_{S'}) \leftarrow S'$
while the sets associated to $\mathcal{V}$ are not disjoint
  $v_1, v_2 \leftarrow \arg\max_{v_1, v_2 \in \mathcal{V}} (|\mathrm{S}(v_1) \cap \mathrm{S}(v_2)|)$
  add the node v_3
  $\mathrm{S}(v_3) \leftarrow \mathrm{S}(v_1) \cap \mathrm{S}(v_2)$
  $\mathrm{S}(v_1) \leftarrow \mathrm{S}(v_1) \setminus \mathrm{S}(v_3)$
  $\mathrm{S}(v_2) \leftarrow \mathrm{S}(v_2) \setminus \mathrm{S}(v_3)$
  add the edges $(v_3, v_1), (v_3, v_2)$
for $v \in \mathcal{V}$
  for $n \in \mathrm{S}(v)$
    add the node v_n
    add the edge (v_n, v)
    $\mathrm{S}(v) \leftarrow \mathrm{S}(v) \setminus \{n\}$
compute the weight of each node as the number of sources below it
for every node with indegree > 1
  build a Huffman tree from the parents to the node
return $\mathcal{G}$

Figure 4.3: Algorithm 1

Let $\mathcal{G}_{\mathrm{lat}} = (\mathcal{V}_{\mathrm{lat}}, \mathcal{E}_{\mathrm{lat}})$ denote the resulting lattice graph. In the second phase, for every node $v \in \mathcal{V}_{\mathrm{lat}}$ for all $n \in \mathrm{S}(v)$, a source v_n representing user n together with edge (v_n, v') is added to the graph. Finally, for every node v with $\mathrm{indeg}(v) \geq 3$, a Huffman tree from the parents to the node is built. Here, the weight of a source is 1, and the weight of non-sources is given as the number of sources below it.

Properties of the lattice graph. We now derive several properties of the lattice graph, which will be used to prove correctness and compute bounds on the total update cost of the generated key-derivation graph. Thus, let $\mathcal{G}_{\mathrm{lat}} = (\mathcal{V}_{\mathrm{lat}}, \mathcal{E}_{\mathrm{lat}})$ be the lattice graph generated on input of $[N]$ and set of k groups $\mathcal{S} = \{S_1, \ldots, S_k\} \subseteq 2^{[N]}$. For

index set $I' \subseteq [k]$ we denoted by

$$P_{I'} \coloneqq \bigcap_{i \in I'} S_i \setminus \bigcup_{j \in [k] \setminus I'} S_j \ ,$$

the set of parties that are members of exactly the groups specified by I. Further, for $v \in \mathcal{V}_{\text{lat}}$ we define

$$I(v) \coloneqq \{i \in [k] \mid \text{exists path from } v \text{ to } v_{S_i}\} \ ,$$

the index set of group nodes that can be reached from v. Finally, for a collection $\mathcal{V}' \subseteq \mathcal{V}$ of nodes we generalize the notation for associated sets to $\mathrm{S}(\mathcal{V}') := \cup_{v \in \mathcal{V}'} \mathrm{S}(v)$. We obtain the following.

Lemma 4.5.1. *Let $N, k \in \mathbb{N}$, $\mathcal{S} = \{S_1, \ldots, S_k\} \subseteq 2^{[N]}$, and let $\mathcal{G}_{\text{lat}} = (\mathcal{V}_{\text{lat}}, \mathcal{E}_{\text{lat}})$ be the lattice graph generated on input $([N], \mathcal{S})$. Then the following holds.*

1. *Let $v, v' \in \mathcal{V}_{\text{lat}}$ be such that $I(v) = I(v')$. Then $v = v'$.*
2. *$I(v) \neq \emptyset$ for all $v \in \mathcal{V}_{\text{lat}}$.*
3. *For every $v \in \mathcal{V}_{\text{lat}}$ and every $i \in I(v)$ there is exactly one path from v to v_{S_i}.*
4. *Consider the ancestor graph $\mathcal{A}(v)$ for $v \in \mathcal{V}_{\text{lat}}$. Then*

$$\bigcup_{v' \in \mathcal{A}(v)} \mathrm{S}(v') \subseteq \bigcap_{i \in I(v)} S_i \ .$$

   If $|I(v)| = 1$ then the equation holds with equality, i.e., $\bigcup_{v' \in \mathcal{A}(v_S)} \mathrm{S}(v') = S$ for all $S \in \mathcal{S}$.

5. *Consider some $v \in \mathcal{V}_{\text{lat}}$. Then we have $\mathrm{S}(v) = P_{I(v)}$.*

Before turning to the proof, we briefly discuss how Lemma 4.5.1 allows us to interpret the lattice graph as a subgraph of the Boolean lattice with respect to the power set of $[k]$, i.e., the graph $\mathcal{G}_B = (\mathcal{V}_B, \mathcal{E}_B)$ with $\mathcal{V}_B = \{v_I \mid I \subseteq [k]\}$ and edges $\mathcal{E}_B = \{(v_I, v_{I'}) \mid I, I' \subseteq [k] : I' \subseteq I)\}$. Indeed, Properties 1 and 2 allow us to map every $v \in \mathcal{V}_{\text{lat}}$ to a unique index set $I \subseteq [k]$. Since the existence of an edge $(v, v') \in \mathcal{E}_{\text{lat}}$ implies that $I(v) \supseteq I(v')$ all edges adhere to the structure of $\mathcal{G}_B$. Summing up, the map $\mathcal{G} \to \mathcal{G}_B; v \mapsto v_{I(v)}$ is an injective graph homomorphism. This allows us to identify nodes of the lattice graph with nodes of $\mathcal{G}_B$ and sometimes write $v_{I'}$ for a unique node $v \in \mathcal{V}_{\text{lat}}$ with $I(v) = I' \in \mathcal{P}([k])$. By Property 5 the associated set of v is P_I, the set of users exactly in the groups specified by I. Figure 4.4 depicts an example execution of Algorithm 1.

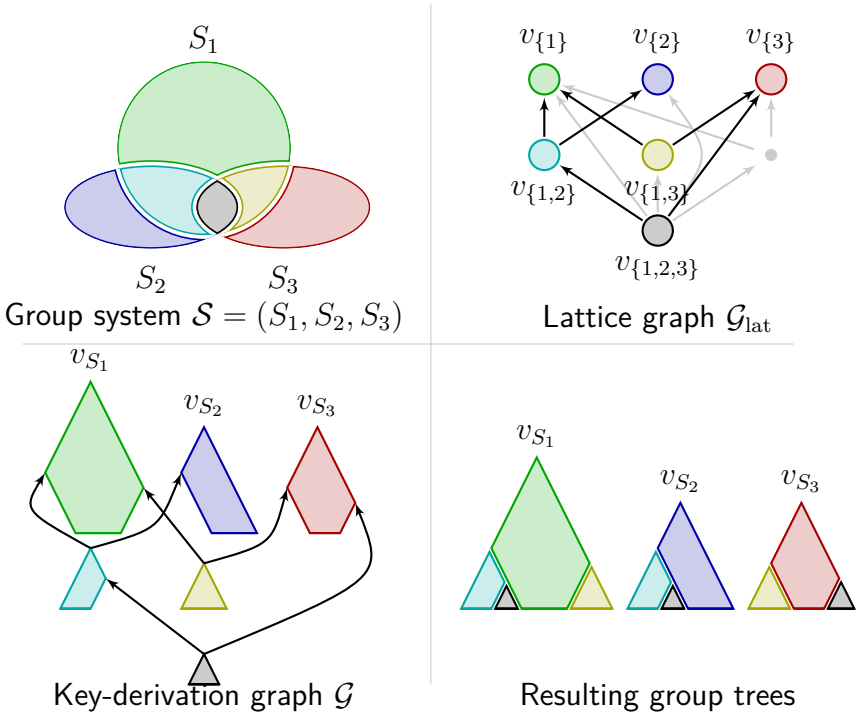

Figure 4.4: Working principle of the algorithm. Top left; Venn diagram of the considered group system. Top right; resulting lattice graph after the first phase. Node v_I has associated set $\mathrm{S}(v_I) = P_I$, the set of users in exactly the groups indicated by I. Nodes and edges of the Boolean lattice that are not part of $\mathcal{G}_{\text{lat}}$ are depicted in gray. Bottom left; final key derivation graph. Bottom right; resulting trees corresponding to groups S_1, S_2, S_3. Note that components of the same color are shared among different trees.

Proof of Lemma 4.5.1. For $t \in \mathbb{N}$ let $\mathcal{G}_t = (\mathcal{V}_t, \mathcal{E}_t)$ be the graph after the tth execution of the loop **while** loop of line 5. We will further use S_t and $\mathcal{A}_t$ to denote associated sets and ancestor sets with respect to $\mathcal{G}_t$.

We show via induction on t that Properties 2 to 4 hold in $\mathcal{G}_t$ and that additionally for all $v \in \mathcal{V}$ the sets associated to $\mathcal{A}_t(v)$ are pairwise disjoint. For $t = 0$ the graph consists of isolated nodes $v_{S_1}, \ldots, v_{S_k}$ with associated sets $S_1, \ldots, S_k$ and corresponding index sets $\{1\}, \ldots, \{k\}$. Thus, all properties stated above clearly hold.

Assume that the properties hold for all steps up to $t-1$ and consider $\mathcal{G}_t$. The only change to $\mathcal{E}$ in the tth step is the addition of two new edges (v_3, v_1) and (v_3, v_2). Thus, by the induction hypothesis we have $I(v) \neq \emptyset$ for all $v \in \mathcal{V}_t \setminus \{v_3\}$. Further $I(v_3) = I(v_1) \cup I(v_2) \neq \emptyset$ and Property 2 still holds.

Regarding Property 3 note that the new node v_3 added in the tth execution of the

loop is a source. Thus, by the induction hypothesis for all $v \neq v_3$ there still exists exactly one path to every corresponding v_{S_i}. We have $I(v_3) = I(v_1) \cup I(v_2)$. Since v_3 is connected to both v_1 and v_2, by the induction hypothesis there is at least one path from v_3 to v_{S_i} for all $i \in I(v_3)$. Now assume that there is an i such that there are at least two different paths from v_3 to v_{S_i}. By the induction hypothesis these paths must diverge in v_3, i.e., there are paths from v_1 to v_{s_i} and v_2 to v_{S_i}. But since v_1 and v_2 were chosen by the algorithm in the tth execution of the loop this implies that after the $(t-1)$th execution there were two ancestors of v_{S_i} with non-disjoint associated sets. A contradiction to the induction hypothesis. Thus Property 3 holds.

Regarding Property 4 consider $v \in \mathcal{V}_t$. Note that we either have $\mathcal{A}_t(v) = \mathcal{A}_{t-1}(v)$ or $\mathcal{A}_t(v) = \mathcal{A}_{t-1}(v) \cup \{v_3\}$, v_3 being the newly added node. In the former case Property 4 and disjointness of associated ancestor sets follow immediately from the induction hypothesis. For the latter, first consider the case $v \neq v_3$. Note that the index set $I(v)$ of v remains unchanged. Assume without loss of generality that for the two nodes v_1, v_2 processed by the algorithm in the tth execution of the loop we have $v_1 \in \mathcal{A}_{t-1}(v)$. Then the associated set of v_1 is updated to $\mathrm{S}_t(v_1) = \mathrm{S}_{t-1}(v_1) \setminus \mathrm{S}_{t-1}(v_2)$, $\mathrm{S}_t(v_3) = \mathrm{S}_{t-1}(v_1) \cap \mathrm{S}_{t-1}(v_2)$, and all other associated sets stay unchanged. Thus all sets are still pairwise disjoint and cover the same subset of $\bigcap_{i \in I(v)} S_i$. In particular if $|I(v)| = 1$ we still have $\bigcup_{v \in \mathcal{A}(v)} \mathrm{S}(v) = \bigcap_{i \in I(v)} S_i$. Now consider v_3. We have $I(v_3) = I(v_1) \cup I(v_2)$ which in particular implies $|I(v_3)| \geq 2$. By the induction hypothesis we obtain that

$$\mathrm{S}_t(v_3) = \mathrm{S}_{t-1}(v_1) \cap \mathrm{S}_{t-1}(v_2) \subseteq \Big(\bigcap_{i \in I(v_1)} S_i\Big) \cap \Big(\bigcap_{i \in I(v_2)} S_i\Big) = \bigcap_{i \in I(v_3)} S_i \ .$$

This concludes the induction.

We now prove Property 1. Note that $|I(v)| = 1$ implies $v = v_{S_i}$ for some index i. In this case the property holds by construction. Thus let $v, v' \in \mathcal{V}_{\mathrm{lat}}$ be such that $I(v) = I(v')$ and $|I(v)| \geq 2$. Consider the sets $\mathcal{D}(v)$ and $\mathcal{D}(v')$ of descendants of v and v' respectively. We first show that either $v \in \mathcal{D}(v')$ or $v' \in \mathcal{D}(v)$. To this end, note that by Property 3 the set $M \coloneqq \{v_{S_i} \mid i \in I(v)\}$ is a subset of $\mathcal{D}(v) \cap \mathcal{D}(v')$. Let $v_1, v_2 \in M$ be such that v_3 was the first node of $\mathcal{D}(v) \cup \mathcal{D}(v')$ added by the algorithm together with the corresponding edges $(v_3, v_1), (v_3, v_2)$, and let t be the point in time when v_3 was added.

Assume without loss of generality that $v_3 \in \mathcal{D}(v)$. We show that $v_3 \in \mathcal{D}(v')$ holds as well. Since $I(v) = I(v')$ there must exist paths from v' to v_1 and v_2. Let v_1^* and v_2^* be the parents of v_1 and v_2 on those paths respectively, and let t_1 and t_2 denote the points in time when v_1^* and v_2^* where added to the graph. Note that by choice of v_3 all nodes that were added before v_3 must have a path to some v_{S_i} with $i \notin I(v)$ and hence by Property 3 cannot be elements of $\mathcal{D}(v')$, which implies that both $t_1 > t$ and

$t_2 > t$. If $v_1^* \neq v_3 \neq v_2^*$ then we have

$$\mathrm{S}_{t_1}(v_1^*) \subseteq \mathrm{S}_{t-1}(v_1) \setminus \mathrm{S}_{t-1}(v_2) \quad \text{and} \quad \mathrm{S}_{t_2}(v_2^*) \subseteq \mathrm{S}_{t-1}(v_2) \setminus \mathrm{S}_{t-1}(v_1)$$

since v_1^* and v_2^* were added to the graph after v_3. Thus, the nodes' associated sets are disjoint which excludes the possibility of $t_1^* = t_2^*$ and $v_1^* = v_2^*$. Further, the associated sets of v_1^* and v_2^* being disjoint at the time of their creation contradicts that the nodes share an ancestor v' in the lattice graph. we conclude that the paths must go via v_3 and obtain $v_3 \in \mathcal{D}(v')$.

Note that by Property 3 v_3 is the only node in $\mathcal{D}(v)$ and $\mathcal{D}(v')$ that has edges to v_1 or v_2. By replacing M with $(M \cup \{v_3\}) \setminus \{v_1, v_2\}$ we can use the same argument to show that also the node of $\mathcal{D}(v) \cup \mathcal{D}(v')$ added second must be an element of both $\mathcal{D}(v')$ and $\mathcal{D}(v')$. After finitely many steps we either obtain $v \in \mathcal{D}(v')$ or $v' \in \mathcal{D}(v)$.

Assume without loss of generality the former holds, and assume $v \neq v'$. Then there exists a path from v' to v that uses one of the two outgoing edges of v'. Further, the second outgoing edge must be part of a path to some v_{S_i}, where by correctness $i \in I(v')$. Since $I(v) = I(v')$ there also must exist a path from v to v_{S_i}. Thus, there are at least two paths from v' to v_{S_i} contradicting Property 3. We obtain $v = v'$, which concludes the proof of Property 1.

We now prove Property 5. Consider $v \in \mathcal{V}_{\mathrm{lat}}$. As $v \in \mathcal{A}(v)$ we have by Property 4 that $\mathrm{S}(v) \subseteq \bigcap_{i \in I(v)} S_i$. Assume that there is $n \in \mathrm{S}(v)$ such that $n \in S_j$ for some $j \in [k] \setminus I(v)$. By Property 4, we have that $S_j = \bigcup_{v' \in \mathcal{A}(v_{S_j})} \mathrm{S}(v')$ which would imply that $v \in \mathcal{A}(v_{S_j})$. This however contradicts $j \notin I(v)$ and we obtain

$$\mathrm{S}(v_{I'}) \subseteq \bigcap_{i \in I(v)} S_i \setminus \bigcup_{j \in [k] \setminus I(v)} S_j = P_{I(v)} \ .$$

For the other direction consider $n \in P_{I(v)}$ and let $v' \in \mathcal{V}$ be the node such that $n \in \mathrm{S}(v')$. By Property 4, we have that $n \in \bigcup_{u \in \mathcal{A}(v_{S_i})} \mathrm{S}(u)$ for all $i \in I(v)$ and $n \notin \bigcup_{u \in \mathcal{A}(v_{S_j})} \mathrm{S}(u)$ for all $j \in [k] \setminus I(v)$. This implies that $I(v') = I(v)$. By Property 1 we obtain $v = v'$ and in turn $P_{I(v)} \supseteq \mathrm{S}(v)$. □

Correctness. We show that key-derivation graph $\mathcal{G}$ output by Algorithm 1 satisfies the correctness properties of Definition 4.3.1. Note that the first property holds by construction.

To see that the second property holds as well, consider the lattice graph. By Lemma 4.5.1, Property 4 for every group $S' \in \mathcal{S}$ the associated sets of the ancestors of $v_{S'}$ form a partition of S'. In the second phase of the algorithm a source v_n is added for every user and connected to corresponding node in the lattice graph. Thus, after this step the set of users with a path to $v_{S'}$ is exactly S'. As this property remains unaffected by the binarization step of line 19 the final key-derivation graph is indeed correct.

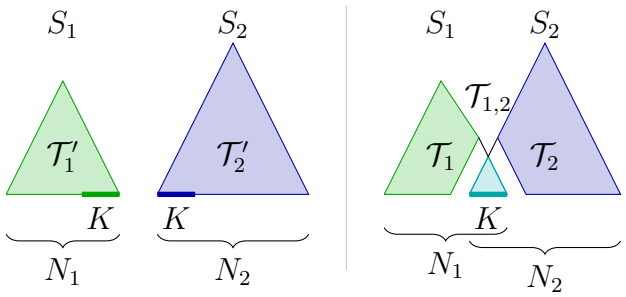

Figure 4.5: Key-derivation graphs of the trivial algorithm (left) and Algorithm 1 (right) for two subgroups. Users that are members of both subgroups are marked in thick.

4.5.2 Total Update Cost

In this section we analyze the total update cost $\mathrm{Upd}(\mathcal{G}) = \sum_{n\in[N]} \mathrm{Upd}(n)$ of key-derivation graphs $\mathcal{G}$ generated by Algorithm 1. To this end, we will split $\mathrm{Upd}(\mathcal{G})$ into the contribution made by the constituting Huffman trees $\mathcal{T}$. Tree $\mathcal{T}$ has a single root and all non-sources in $\mathcal{T}$ have indegree 2. Let $\mathcal{L}(\mathcal{T})$ denote the set of leaves of $\mathcal{T}$. As argued in Lemma 4.3.2, the update cost of a leaf u with respect to $\mathcal{T}$ corresponds to the length $\mathrm{len}(u)$ of its path to the root. Note, however, that leaves of $\mathcal{T}$ may represent more than one user in the key-derivation graph. Indeed, by construction of the algorithm, the weight w_u of u counts the number of leaves in $\mathcal{G}$ below u. Thus, the contribution of Huffman tree $\mathcal{T}$ towards the total update cost of $\mathcal{G}$ is given by $\mathrm{Upd}(\mathcal{T}) = \sum_{u\in\mathcal{L}(\mathcal{T})} w_u \mathrm{len}(u)$. If $U_{\mathcal{T}}$ is the probability distribution that picks $u \in \mathcal{L}(\mathcal{T})$ with probability proportional to its weight w_u, we can express the update cost of $\mathcal{T}$ in terms of the expected length from leaves to the root as

$$\mathrm{Upd}(\mathcal{T}) = \mathbb{E}[\mathrm{len}(U_{\mathcal{T}})] \cdot \sum_{u\in\mathcal{L}(\mathcal{T})} w_u \ . \tag{4.6}$$

We first consider Algorithm 1 for the simplest case of two subgroups and compare it to the trivial algorithm.

Example 4.5.1. *Let $N \in \mathbb{N}$ and let $\mathcal{S}$ consist of two subgroups S_1, S_2 of sizes N_1 and N_2 respectively. Further assume that $|S_1 \cap S_2| = K$. Consider the key derivation graphs generated by the trivial algorithm and Algorithm 1, which in both cases decompose into several Huffman trees. The trivial algorithm essentially generates two trees $\mathcal{T}_1'$ and $\mathcal{T}_2'$, the first containing all members of S_1, the other all members of S_2. Algorithm 1 first collects the K parties that are members of both groups in a tree $\mathcal{T}_{1,2}$. The remaining $(N_1 - K)$ members of S_1 and the root of $\mathcal{T}_{1,2}$ are collected in a tree $\mathcal{T}_1$, the remaining $(N_2 - K)$ members of S_2 and the root of $\mathcal{T}_{1,2}$ in a tree $\mathcal{T}_2$ (See Figure 4.5).*

By Equation 4.6 we have

$$\mathrm{Upd}(\mathcal{G}_{\mathrm{triv}}) = \mathrm{Upd}(\mathcal{T}_1') + \mathrm{Upd}(\mathcal{T}_2') = N_1\, \mathbb{E}[\mathrm{len}(U_{\mathcal{T}_1'})] + N_2\, \mathbb{E}[\mathrm{len}(U_{\mathcal{T}_2'})]$$

and

$$\begin{aligned}\mathrm{Upd}(\mathcal{G}_{\mathrm{a1}}) &= \mathrm{Upd}(\mathcal{T}_1) + \mathrm{Upd}(\mathcal{T}_2) + \mathrm{Upd}(\mathcal{T}_{1,2}) \\ &= N_1\,\mathbb{E}[\mathrm{len}(U_{\mathcal{T}_1})] + N_2\,\mathbb{E}[\mathrm{len}(U_{\mathcal{T}_2})] + K\,\mathbb{E}[\mathrm{len}(U_{\mathcal{T}_{1,2}})]\ .\end{aligned}$$

By optimality of Huffman codes (Lemma 4.2.1) we have that

$$H(U_{\mathcal{T}}) \leq \mathbb{E}[\mathrm{len}(U_{\mathcal{T}})] \leq H(U_{\mathcal{T}}) + 1$$

for $\mathcal{T} \in \{\mathcal{T}_1', \mathcal{T}_2', \mathcal{T}_1, \mathcal{T}_2, \mathcal{T}_{1,2}\}$, where $H(U_{\mathcal{T}})$ is the Shannon entropy of $U_{\mathcal{T}}$. For $\mathcal{T}_1'$, $\mathcal{T}_2'$, and $\mathcal{T}_{1,2}$ the leaves are distributed uniformly and we have $H(\mathcal{T}_1') = \log(N_1)$, $H(\mathcal{T}_2') = \log(N_2)$, $H(\mathcal{T}_{1,2}) = \log(K)$. Let $i \in \{1,2\}$ and consider $\mathcal{T}_i$. Then the first $N_i - K$ leaves have probability $1/N_i$ and the last leaf K/N_i. Thus $H(U_{\mathcal{T}_i}) = (N_i - K)/N_i \log(N_i) + K/N_i \log(N_i/K) = \log(N_i) - K/N_i \log(K)$. Summing up we obtain

$$\begin{aligned}&\mathrm{Upd}(\mathcal{G}_{\mathrm{triv}}) - \mathrm{Upd}(\mathcal{G}_{\mathrm{a1}}) \\ \geq& N_1 \log(N_1) + N_2 \log N_2 - N_1(\log(N_1) - K/N_1 \log(K) + 1) \\ &\quad - N_2(\log(N_2) - K/N_2 \log(K) + 1) - K(\log(K) + 1) \\ =& K(\log(K) - 1) - (N_1 + N_2)\ .\end{aligned}$$

Note that for $K \geq 2$ the first term is non-negative (For $K = 1$ it is easy to see that Algorithm 1 performs better than the trivial algorithm.).

Before turning to arbitrary group systems, we derive a generalized statement on the update cost $\mathrm{Upd}(\mathcal{T})$ contributed by Huffman trees as defined above.

Lemma 4.5.2. *Let $\mathcal{T}$ be a Huffman tree over leaves $v_1, \ldots, v_\ell$ of weight $w_1, \ldots, w_\ell \in \mathbb{N}$. Let $w = \sum_{i=1}^{\ell} w_i$. Then $\mathcal{T}$'s update cost is bounded by*

$$w \log(w) - \sum_{i=1}^{\ell} w_i \log(w_i) \leq \mathrm{Upd}(\mathcal{T}) \leq w(\log(w) + 1) - \sum_{i=1}^{\ell} w_i \log(w_i)\ .$$

Proof. Let $U_{\mathcal{T}}$ denote the probability distribution that picks leaf v_i with probability w_i/w proportional to its weight. The entropy of $U_{\mathcal{T}}$ is given by

$$\begin{aligned}H(U_{\mathcal{T}}) &= -\sum_{i=1}^{\ell} \frac{w_i}{w} \log(w_i/w) = \sum_{i=1}^{\ell} \frac{w_i}{w} \log(w) - \sum_{i=1}^{\ell} \frac{w_i}{w} \log(w_i) \\ &= \log(w) - \sum_{i=1}^{\ell} \frac{w_i}{w} \log(w_i)\ .\end{aligned}$$

The claim follows since by Equation 4.6 the update cost of $\mathcal{T}$ with respect to the weights is given by $\mathbb{E}[\mathrm{len}(U_{\mathcal{T}})] \cdot w$ and by optimality of Huffman codes $H(U_{\mathcal{T}}) \leq \mathbb{E}[\mathrm{len}(U_{\mathcal{T}})] \leq H(U_{\mathcal{T}}) + 1$. □

Regarding general systems of subgroups we obtain the following .

Theorem 4.5.3. *Let $N \in \mathbb{N}$, $S_1, \dots, S_k \subseteq [N]$, and $\mathcal{G}$ the key-derivation graph output by Algorithm 1. Let $\mathcal{G}_{\mathrm{lat}} = (\mathcal{V}_{\mathrm{lat}}, \mathcal{E}_{\mathrm{lat}})$ be the corresponding lattice graph. Then*

$$\sum_{i=1}^{k} |S_i| \cdot \log(|S_i|) - \sum_{v \in \mathcal{V}_{\mathrm{lat}} \colon |I(v)| \geq 2} \left| \bigcup_{v' \in \mathcal{A}(v)} P_{I(v')} \right| \cdot \log\left(\left| \bigcup_{v' \in \mathcal{A}(v)} P_{I(v')} \right| \right) \tag{4.7}$$

$$\leq \mathrm{Upd}(\mathcal{G})$$

$$\leq \sum_{i=1}^{k} |S_i| \cdot (\log(|S_i|) + 1) - \sum_{v \in \mathcal{V}_{\mathrm{lat}} \colon |I(v)| \geq 2} \left| \bigcup_{v' \in \mathcal{A}(v)} P_{I(v')} \right| \cdot \left(\log\left(\left| \bigcup_{v' \in \mathcal{A}(v)} P_{I(v')} \right| \right) - 1 \right) , \tag{4.8}$$

where $\mathcal{A}(v)$ denotes the set of ancestors of v in $\mathcal{G}_{\mathrm{lat}}$, $I(v) = \{i \in [k] \colon \exists$ path from v to $v_{S_i}\}$, and for $I' \subseteq [N]$ the set $P_{I'} \coloneqq \bigcap_{i \in I'} S_i \setminus \bigcup_{j \in [k] \setminus I'} S_j$ indicates the users exactly in the subgroups corresponding to I'.

Proof. As in Example 4.5.1 we decompose the total update cost of $\mathcal{G}$ into parts contributed by Huffman trees. Note that every node $v' \in \mathcal{V}_{\mathrm{lat}}$ of the lattice graph serves as the root of a Huffman tree $\mathcal{T}_{v'}$ in the final key-derivation graph $\mathcal{G}$ and we can compute the total update cost of $\mathcal{G}$ as $\mathrm{Upd}(\mathcal{G}) = \sum_{v' \in \mathcal{V}_{\mathrm{lat}}} \mathrm{Upd}(\mathcal{T}_{v'})$.

The leaves $\mathcal{L}(\mathcal{T}_{v'})$ of $\mathcal{T}_{v'}$ are either sources that were added in the second phase of the algorithm, or nodes that are a parent of v' in the lattice graph.

Recall that the weight of leaves is defined as 1 and for general nodes as the number of leaves below it. This implies that node v'' in line 17 of the algorithm gets assigned weight $|\mathrm{S}(\mathcal{A}(v''))| = \left| \bigcup_{\tilde{v} \in \mathcal{A}(v'')} \mathrm{S}(\tilde{v}) \right|$.

Let $v'_1, \dots, v'_\ell$ denote the parents of v' in $\mathcal{G}_{\mathrm{lat}}$ and consider Huffman tree $\mathcal{T}_{v'}$. Then $\mathcal{T}_{v'}$ has leaves v'_i with weight $w_{v'_i} = |\mathrm{S}(\mathcal{A}(v'_i))|$ for $i \in \{1, \dots, \ell\}$, and $|\mathrm{S}(\mathcal{A}(v'))| - \sum_{i=1}^{\ell} |\mathrm{S}(\mathcal{A}(v'_i))|$ additional leaves, each of which has weight 1.

In the following we will use f as shorthand for the function $f \colon n \mapsto n \log(n)$. We now bound $\mathrm{Upd}(\mathcal{T}_{v'})$ using Lemma 4.5.2. As the negative terms in Lemma 4.5.2's statement contributed by leaves of weight 1 are $1 \cdot \log(1) = 0$ we obtain that

$$\mathrm{Upd}(\mathcal{T}_{v'}) \geq f(|\mathrm{S}(\mathcal{A}(v'))|) - \sum_{i=1}^{\ell} f(|\mathrm{S}(\mathcal{A}(v'_i))|)$$

and

$$\mathrm{Upd}(\mathcal{T}_{v'}) \leq f(|\mathrm{S}(\mathcal{A}(v'))|) + |\mathrm{S}(\mathcal{A}(v'))| - \sum_{i=1}^{\ell} f(|\mathrm{S}(\mathcal{A}(v'_i))|) \ .$$

To compute the total update cost of $\mathcal{G}$ we have to sum over all trees $\mathcal{T}_{v'}$ with $v' \in \mathcal{V}_{\text{lat}}$. Note that every node v' in $\mathcal{V}_{\text{lat}}$ with $|I(v')| \geq 2$ has outdegree 2. Thus if we sum over the update cost of all trees the term $f(|\mathrm{S}(\mathcal{A}(v'))|$ appears twice with a negative sign (in the cost of v''s children) and once with a positive (in $\mathrm{Upd}(\mathcal{T}_{v'})$). Thus

$$\begin{aligned}
&\sum_{v' \in \mathcal{V}_{\text{lat}} \colon |I(v')|=1} f(|\mathrm{S}(\mathcal{A}(v'))|) - \sum_{v' \in \mathcal{V}_{\text{lat}} \colon |I(v')|\geq 2} f(|\mathrm{S}(\mathcal{A}(v'))|) \\
\leq &\mathrm{Upd}(\mathcal{G}) = \sum_{v' \in \mathcal{V}_{\text{lat}}} \mathrm{Upd}(\mathcal{T}_{v'}) \\
\leq &\sum_{v' \in \mathcal{V}_{\text{lat}} \colon |I(v')|=1} (f(|\mathrm{S}(\mathcal{A}(v'))|) + |\mathrm{S}(\mathcal{A}(v'))|) \\
&\quad - \sum_{v' \in \mathcal{V}_{\text{lat}} \colon |I(v')|\geq 2} \Big(f(|\mathrm{S}(\mathcal{A}(v'))|) - |\mathrm{S}(\mathcal{A}(v'))|\Big) \ .
\end{aligned}$$

This is equivalent to the claim of the theorem since the nodes v' with $|I(v')| = 1$ are exactly the group-key nodes of the form $v' = v_{S_i}$ and since by correctness of the algorithm $|\mathrm{S}(\mathcal{A}(v_{S_i}))| = |S_i|$ and by Lemma 4.5.1, Property 5 $\mathrm{S}(v_{I'}) = P_{I'}$ where the partitions $P_{I'}$ are disjoint. □

The bounds of Theorem 4.5.3 depend on the structure of the lattice graph generated by the algorithm. Using Properties 4 and 5 of Lemma 4.5.1 to bound $|\mathrm{S}(\mathcal{A}(v'))|$ it is possible to obtain a weaker bound on $\mathrm{Upd}(\mathcal{G})$ that only depends on $[N]$ and $\mathcal{S}$.

We conclude the section by comparing the update cost of Algorithm 1 to that of the trivial algorithm and the asymptotically optimal algorithm of Section 4.4.1.

Comparison to the trivial algorithm. Note that the terms $\sum_{i=1}^{k} |S_i| \cdot \log(|S_i|)$ and $\sum_{i=1}^{k} |S_i| \cdot (\log(|S_i|) + 1)$ in Theorem 4.5.3 match the bounds on the update cost of the trivial algorithm derived in Section 4.3.4. Thus the second term of

$$\sum_{v \in \mathcal{V}_{\text{lat}} \colon |I(v)|\geq 2} \Big| \bigcup_{v' \in \mathcal{A}(v)} P_{I(v')} \Big| \cdot \left(\log \left(\Big| \bigcup_{v' \in \mathcal{A}(v)} P_{I(v')} \Big| \right) - 1 \right)$$

provides a good estimate on how much Algorithm 1 saves compared to the trivial one. For the group system depicted in Figure 4.4, for example, this would amount to

$$|S_1 \cap S_2| \cdot \log(|S_1 \cap S_2|) + |S_1 \cap S_3 \setminus S_2| \cdot \log(|S_1 \cap S_3 \setminus S_2|) + |S_1 \cap S_2 \setminus S_3| \cdot \log(|S_1 \cap S_2 \cap S_3|) \ .$$

Due to the "rounding error" of $+1$ in $\sum_{i=1}^{k} |S_i| \cdot (\log(|S_i|) + 1)$, Theorem 4.5.3 unfortunately does not allow us to conclude that the update cost of Algorithm 1 always improves on the one of the trivial algorithm. In Appendix 4.8, we provide an alternative analysis of $\mathrm{Upd}(\mathcal{G})$ that directly compares the two algorithms and gives conditions that imply Algorithm 1 outperforming the trivial one.

Comparison to the asymptotically optimal algorithm of Section 4.4.1. The algorithm of Section 4.4.1 in a first step constructs a binary tree for every non-empty partition $P_{I'}$ and then, in a second step, builds a binary tree for every group using the roots of the "partition trees" as leafs. We can interpret this as an algorithm that, similarly to Algorithm 1, in the first phase chooses a lattice graph $\mathcal{G}_{\text{lat}}$, concretely the graph that connects every node $v_{I'}$ directly with edges to all corresponding group nodes $\{v_{\{i\}} \mid i \in I'\}$, and in the second phase builds Huffman trees for every lattice node.[7]

Thus, by Lemma 4.5.2, we can lower bound the update cost of key graphs $\mathcal{G}_{\text{asopt}}$ generated by it by

$$\text{Upd}(\mathcal{G}_{\text{asopt}}) \geq \sum_{i=1}^{k} \Big(|S_i| \cdot \log(|S_i|) - \sum_{I' \subseteq [N] : i \in I' \wedge |I'| \geq 2} |P_{I'}| \cdot \log(|P_{I'}|) \Big) + \sum_{I' \subseteq [N] : |I'| \geq 2} |P_{I'}| \cdot \log(|P_{I'}|) \ ,$$

which, taking into account that every I' with $|I'| = \ell$ corresponds to exactly ℓ groups, simplifies to

$$\text{Upd}(\mathcal{G}_{\text{asopt}}) \geq \sum_{i=1}^{k} |S_i| \cdot \log(|S_i|) - \sum_{I' \subseteq [N] : |I'| \geq 2} (|I'| - 1) |P_{I'}| \cdot \log(|P_{I'}|) \ . \tag{4.9}$$

For a comparison to Algorithm 1, consider a key derivation graph $\mathcal{G}_{\text{a1}}$ output by it. We now compute a lower bound on $\text{Upd}(\mathcal{G}_{\text{asopt}}) - \text{Upd}(\mathcal{G}_{\text{a1}})$. Let $\mathcal{G}'_{\text{lat}}$ be the lattice graph of $\mathcal{G}_{\text{a1}}$ and $v_{I'} \in \mathcal{G}'_{\text{lat}}$ such that $|I'| \geq 2$. Every non-sink in $\mathcal{G}'_{\text{lat}}$ has outdegree 2 and $v_{I'}$ is connected to all $v_{\{i\}}$ with $i \in I'$ by exactly one path. Thus, the subgraph of $\mathcal{G}'_{\text{lat}}$ induced by these paths is a binary tree with root $v_{I'}$ and $|I'|$ leafs, and thus consists of exactly $2|I'| - 1$ nodes, $|I'|$ of which have an index set of size 1. This implies that there exists $|I'| - 1$ many nodes $v_{I''}$ in $\mathcal{G}'_{\text{lat}}$ with $|I''| \geq 2$ such that $v_{I'} \in \mathcal{A}(v_{I''})$.

Using f as shorthand for the function $f : N \mapsto N \log(N)$ and $p_{I'} = |P_{I'}|$, we now can distribute the expressions $|P_{I'}| \cdot \log(|P_{I'}|)$ of Equation 4.9 on the negative summands of Equation 4.8 and obtain

$$\text{Upd}(\mathcal{G}_{\text{asopt}}) - \text{Upd}(\mathcal{G}_{\text{a1}}) \geq \sum_{v \in \mathcal{V}'_{\text{lat}} : |I(v)| \geq 2} \Big(f\Big(\sum_{v' \in \mathcal{A}(v)} p_{I(v')} \Big) - \sum_{v' \in \mathcal{A}(v)} f(p_{I(v')}) - 1 \Big) \ .$$

Note that the function f grows super-linearly implying that the terms $f(\sum_{v' \in \mathcal{A}(v)} p_{I(v')}) - \sum_{v' \in \mathcal{A}(v)} f(p_{I(v')})$ are non-negative, and can even be of order N as for example $f(2N/2) - 2f(N/2) = N$. While, again due to the terms -1, we are unfortunately not able to conclude that Algorithm 1 is always more efficient, this shows that it still can save substantially in terms of update cost, in particular if the $p_{I'}$ are large.

[7] Formally, the algorithm as described in Section 4.4.1 collects all users that are *only* in group S_i in a tree before computing the tree for S_i, while in the lattice-graph variant these users are directly included in the tree for S_i. Note, however, that the latter approach can only improve the total update cost.

4.5.3 Maximal Update Cost per User

In the previous section we were considering the total update cost of key-derivation graphs generated by Algorithm 1, which relates to the average update cost of parties. As we have shown this metric will typically improve compared to the trivial algorithm. However, it might still be possible, that the update cost of particular, fixed users increases. In this section we show that while this may indeed happen, the increase is essentially bounded by a small constant.

As Algorithm 1 builds on Huffman codes, the results of this section make use of weight distributions that maximize the codeword length of such codes, concretely, weights corresponding to the Fibonacci numbers F_i that are recursively defined by

$$F_0 = 0, F_1 = 1 \quad \text{and} \quad F_i = F_{i-1} + F_{i-2} \text{ for } i \geq 2 \ .$$

We will make use of the following facts from [AMM00]:

$$\sum_{i=1}^{k} F_i = F_{k+2} - 1 \tag{4.10}$$

$$k - 2 < \log_{\Phi}(F_k) < k - 1 \ , \tag{4.11}$$

where $\Phi = (1 + \sqrt{5})/2$.

We first consider an example in which the update cost of a fixed user increases compared to the trivial solution.

Example 4.5.2. *Recall that for a system of subgroups $\mathcal{S} = \{S_1, \cdots, S_k\}$ the set of parties exactly in the subgroups specified by index set $I \subseteq [k]$ is given by $P_I = \bigcap_{i \in I} S_i \setminus \bigcup_{j \in [k] \setminus I} S_j$. Now assume that $\mathcal{S}$ satisfies*

$$\left|P_{\{1\}}\right| = 1, \quad \left|P_{\{1,i\}}\right| = F_i \ \forall i \in \{2, \cdots, k\}, \text{ and } \quad |P_I| = 0 \text{ for all other } I \ .$$

We are interested in the update cost of the single party $n \in P_{\{1\}}$. Since by choice of the P_I we have that $|S_1| = \sum_{i=1}^{k} F_i$, we obtain by Equations 4.10 and 4.11 that n's update cost with respect to the trivial algorithm is maximally $\mathrm{Upd}_{\mathrm{triv}}(n) \leq \lceil (k+1)\log(\Phi) \rceil$.

Now consider n's update cost in a key-derivation graph $\mathcal{G}$ generated by our algorithm. Then v_n is a leaf of weight 1 in the Huffman tree with root v_{S_1}, while the other leaves $k-1$ have weights corresponding to the Fibonacci sequence. Thus, the length of v_n's path to the root and in turn her update cost $\mathrm{Upd}(n)$ is $k-1$.

Summing up, for the considered set system the maximal update cost with respect to Algorithm 1 is larger by a factor of roughly $1/\log(\Phi) \approx 1.44$ compared to the one of the trivial algorithm.

Below we show that the behavior exhibited in the example above is essentially the worst case. We first recall a fact about the maximal length of paths in Huffman trees that follows from plugging Equation 4.11 into [AMM00, Theorem 5].

Fact 1. *Let $\mathcal{T}$ be a Huffman tree over leaves $v_1, \ldots, v_\ell$ of weight $w_1, \ldots, w_\ell \in \mathbb{N}$. Let $w = \sum_{i=1}^{\ell} w_i$. Then for all $j \in \{1, \ldots, \ell\}$ the length of the path from w_j to the root of $\mathcal{T}$ is bounded by*

$$\mathrm{len}(w_j) \leq \log_\Phi(w) - \log_\Phi(w_j) + 1 \ ,$$

where $\Phi = (1 + \sqrt{5})/2$.

Lemma 4.5.4. *Let $N \in \mathbb{N}$, $\mathcal{S} = \{S_1, \ldots, S_k\} \subseteq 2^{[N]}$, and $\mathcal{G}$ the key-derivation graph output by Algorithm 1. Fix a party $n \in [N]$ and let $I' =:= \{i \in [k] \mid n \in S_i\}$. Then n's update cost in $\mathcal{G}$ is bounded from above by*

$$\mathrm{Upd}(n) \leq \sum_{i \in I'} \Big(\lceil \log_\Phi(|S_i|) \rceil + |I'| - 1 \Big) \approx \sum_{i \in I'} \Big(\lceil 1.44 \cdot \log(|S_i|) \rceil + |I'| - 1 \Big) \ .$$

Proof. Let $\mathcal{G}_{\mathrm{lat}} = (\mathcal{V}_{\mathrm{lat}}, \mathcal{E}_{\mathrm{lat}})$ denote the lattice graph corresponding to $\mathcal{G}$. By Lemma 4.5.1, Property 5 the node $v'_0 \in \mathcal{V}_{\mathrm{lat}}$ that the source v_n is connected to in the second phase of the algorithm satisfies $I(v'_0) = I'$. Fix $i \in I'$. By Property 3 v'_0 is connected to group node v_{S_i} by exactly one path $(v'_0, \cdots, v'_\ell = v_{S_i})$. Since $I(v'_0) = I'$ and $I(v'_{j-1}) \supsetneq I(v'_j)$ for all j we have $\ell \leq |I'| - 1$. In the key-derivation graph v'_j is connected to v'_{j-1} by a path P_j for $j \in \{1, \ldots, \ell\}$ and party n's source v_n is connected to v'_0 by a path P_0.

Since the weight of v'_j is $\big|\mathrm{S}(\mathcal{A}(v'_j))\big|$ we obtain by Fact 1 that

$$\mathrm{len}(P_j) \leq \Big\lceil \log_\Phi(\big|\mathrm{S}(\mathcal{A}(v'_j))\big|) - \log_\Phi(\big|\mathrm{S}(\mathcal{A}(v'_{j-1}))\big|) \Big\rceil$$

and that $\mathrm{len}(P_0) \leq \lceil \log_\Phi(|\mathrm{S}(\mathcal{A}(v'_0))|) \rceil$ since v_n has weight 1. Using that $\lceil a - b \rceil + \lceil b \rceil \leq \lceil a \rceil + 1$ for $b, a - b \geq 0$ it follows that

$$\begin{aligned}
\sum_{j=0}^{\ell} \mathrm{len}(P_j) &\leq \lceil \log_\Phi(|\mathrm{S}(\mathcal{A}(v'_0))|) \rceil + \sum_{j=1}^{\ell} \Big\lceil \log_\Phi(\big|\mathrm{S}(\mathcal{A}(v'_j))\big|) - \log_\Phi(\big|\mathrm{S}(\mathcal{A}(v'_{j-1}))\big|) \Big\rceil \\
&\leq \lceil \log_\Phi(|\mathrm{S}(\mathcal{A}(v_\ell))|) \rceil + \ell \\
&\leq \lceil \log_\Phi(|\mathrm{S}(\mathcal{A}(v_\ell))|) \rceil + |I'| - 1 \ .
\end{aligned}$$

Summing over all $i \in I'$ and taking into account that by Lemma 4.5.1, Property 4 $\mathrm{S}(\mathcal{A}(v_{S_i})) = S_i$ yields the claim of the lemma.

□

Note that in the analysis above the component of n's update cost contributed by the Huffman tree rooted at lattice node v_I is included in the bound of the update cost of all S_i with $i \in I$, and thus overestimated by a factor of $|I|$. Thus, the actual update cost of users (in particular if they are members of many groups) will typically be better than Lemma 4.5.4 indicates.

4.5.4 Asymptotic Optimality of Boolean-lattice based Graphs

As discussed in Section 4.5.1, we can interpret our algorithm as follows. On input $([N], \mathcal{S} = \{S_1, \ldots, S_k\})$, in the first phase the algorithm picks a subgraph of the Boolean lattice $\mathcal{G}_B = (\mathcal{V}_B, \mathcal{E}_B)$ with respect to the power set of $[k]$, where

$$\mathcal{V}_B = \{v_I \mid I \subseteq [k]\} \quad \text{and} \quad \mathcal{E}_B = \{(v_I, v_{I'}) \mid I, I' \subseteq [k] : I' \subseteq I)\} \ .$$

We refer to this subgraph as the lattice graph. In the second phase, for $I \subseteq [k]$, a source for every party in P_I, i.e., the set of parties belonging exactly to the groups specified by I, is added and connected to node v_I. Each node in the graph is assigned a weight; sources have weight 1 and the weight of all other nodes is the sum of the weights of their parents. Finally, for every v_I a Huffman tree to its parents according to the weight distribution is built, resulting in the key-derivation graph.

In this section we consider key-derivation graphs for general choices of the lattice graph, i.e., key derivation graphs $\mathcal{G}$ obtained by executing the second phase of the algorithm as described above with respect to a lattice-graph $\mathcal{G}_{\text{lat}} = (\mathcal{V}_{\text{lat}}, \mathcal{E}_{\text{lat}}) \subseteq \mathcal{G}_B$.[8] We say $\mathcal{G}$ is the key-derivation graph associated to $\mathcal{G}_{\text{lat}}$, $[N]$ and $\mathcal{S}$. The following theorem shows that the update cost of every lattice-based key derivation graphs, and in particular graphs generated by Algorithm 1, is optimal in the asymptotic setting of Section 4.4.

Theorem 4.5.5. *Let $k \in \mathbb{N}$ be fixed, and for $I \subseteq [k]$ let $p_I \in [0,1]$ be such that $\sum_{I \subseteq [k]} p_I = 1$ and $p_\emptyset = 0$. For $N \in \mathbb{N}$ let $\mathcal{S}(N)$ be the subgroup system associated to the p_I.*

Let $\mathcal{G}_{\text{lat}} = (\mathcal{V}_{\text{lat}}, \mathcal{E}_{\text{lat}})$ be a subgraph of the Boolean-lattice graph with respect to $[k]$ satisfying that $v_I \in \mathcal{V}_{\text{lat}}$ for all I with $p_I > 0$, and let $\mathcal{G}(N)$ be the key-derivation graph associated to $\mathcal{G}_{\text{lat}}$ and $\mathcal{S}(N)$. Then

$$\text{Upd}(\mathcal{G}(N)) \xrightarrow{N \to \infty} \sum_{I \subseteq [k]} |N \cdot p_I| \cdot \log(|N \cdot p_I|) + \Theta(N) = N \log(N) + \Theta(N) \ .$$

Proof. For node $v_I \in \mathcal{V}_{\text{lat}}$ let $\mathcal{T}_{v_I}^N$ denote the Huffman-tree in $\mathcal{G}(N)$ rooted at v_I. Further, let $v_1, \cdots, v_\ell$ be the parents of v_I in $\mathcal{G}_{\text{lat}}$. We will analyze the contribution $\mathcal{T}_{v_I}^N$ makes to the total update cost of $\mathcal{G}(N)$.

[8] Naturally, one would require that the resulting key-derivation graph satisfies correctness. However, this is not necessary for our analysis of its update cost.

First, we show that if $p_I > 0$ then

$$\mathrm{Upd}(\mathcal{T}_{v_I}^N) \xrightarrow{N\to\infty} Np_I \cdot \log(Np_I) \ .$$

To this end, for $i \in \{1, \ldots, \ell\}$ let q_i be such that the weight of v_i in $\mathcal{G}(N)$ is given by $q_i N$. Since $\mathcal{T}_{v_I}^N$ has $p_I \cdot N$ leaves of weight 1 additional to $v_1, \ldots, v_\ell$ this implies that the weight of v_I in $\mathcal{G}(N)$ is $N \cdot (p_I + \sum_{i=1}^{\ell} q_i)$. We now bound $\mathrm{Upd}(\mathcal{T}_v^N)$ using Lemma 4.5.2. As the negative terms contributed by leaves of weight 1 are $1 \cdot \log(1) = 0$ we obtain that

$$\begin{aligned}
&\mathrm{Upd}(\mathcal{T}_{v_I}^N) \\
\leq& N\Big((p_I + \sum_{i=1}^{\ell} q_i) \cdot \log\left(N(p + \sum_{i=1}^{\ell} q_i)\right) - \sum_{i=1}^{\ell} q_i \log(N \cdot q_i)\Big) \\
=& N\Big(p_I \log(N) + \sum_{i=1}^{\ell} q_i \log(N) + p_I \log(p_I + \sum_{i=1}^{\ell} q_i) + \sum_{i=1}^{\ell} q_i \log(p_I + \sum_{i=1}^{\ell} q_i) \\
&\quad - \sum_{i=1}^{\ell} q_i \log(N) - \sum_{i=1}^{\ell} q_i \log(q_i)\Big) \ ,
\end{aligned}$$

Note that the terms $\sum_{i=1}^{\ell} q_i \log(N)$ cancel out and that $c \coloneqq p_I \log(p_I + \sum_{i=1}^{\ell} q_i) + \sum_{i=1}^{\ell} q_i \log(p_I + \sum_{i=1}^{\ell} q_i) - \sum_{i=1}^{\ell} q_i \log(q_i)$ is independent of N. We thus have

$$\mathrm{Upd}(\mathcal{T}_{v_I}^N) \leq N \cdot (p_I \log(N) + c)$$

and obtain in the case $p > 0$ that

$$1 \leq \frac{\mathrm{Upd}(\mathcal{T}_{v_I})}{Np_I \cdot \log(Np_I)} \leq \frac{p_I \log(N) + c}{p_I \log(N) + p_I \log(p_I)} \ ,$$

where the first inequality is due to Lemma 4.5.2 and the last term converges to 1 as claimed.

Now consider the case $p_I = 0$. In this case the Huffman tree $\mathcal{T}_{v_I}^N$ for all N has exactly ℓ leaves, the proportional weight of which stays unchanged. Thus $\mathcal{T}_{v_I}^N$ is the same for all N and in particular has constant average update size. Recall that by Equation 4.6 the update cost of $\mathcal{T}_{v_I}^N$ is given by

$$\mathrm{Upd}(\mathcal{T}_{v_I}^N) = \mathbb{E}[\mathrm{len}(U_{\mathcal{T}_{v_I}^N})] \cdot w_{v_I} \ .$$

Since $\mathbb{E}[\mathrm{len}(U_{\mathcal{T}_{v_I}^N})]$ as argued above is constant, and since all weights w_{v_I} are linear in N we obtain that $\mathrm{Upd}(\mathcal{T}_{v_I}^N) \in O(N)$.

Summing over all Huffman trees yields the claim of the theorem. □

4.6 Dynamic Operations

So far we considered the setting of systems $\mathcal{S}$ of *static* groups for a universe of users $[N]$, i.e., while the keys in the key-derivation graph are rotated, the set of groups that a particular party is a member of stays unchanged. Naturally, we would like to be able to add or remove users from groups. In this section, we first analyze what these operations correspond to with respect to boolean lattice based key-derivation graphs and then discuss how techniques for adds and removes in CGKA and Multicast schemes in the single-group setting can be adapted to multiple groups.

Dynamic operations with respect to key-derivation graphs. Let $N \in \mathbb{N}$, $\mathcal{S} = \{S_1, \ldots, S_k\} \subseteq 2^{[N]}$, and let $\mathcal{G}$ be a key-derivation graph generated by our algorithm on input $(N, \mathcal{S})$. Further, let $\mathcal{G}_{\text{lat}} = (\mathcal{V}_{\text{lat}}, \mathcal{E}_{\text{lat}})$ be the corresponding lattice graph. Consider a party $n \in [N]$ with index set $I = I(n) = \{i \in [k] \mid n \in S_i\}$. As discussed in Section 4.5 n's node is a leaf of the Huffman tree rooted at $v_I \in \mathcal{V}_{\text{lat}}$. Our goal is to support operations $\mathrm{Add}(n, i)$ which refreshes group key sk_{S_i} and gives n access to it, and $\mathrm{Rem}(n, i)$ which removes n from group i, i.e., replaces sk_{S_i} with a key not known to n.

Conceptually, $\mathrm{Add}(n, i)$ and $\mathrm{Rem}(n, i)$ correspond to changing n's index set I to $I' = I \cup \{i\}$ or $I' = I \setminus \{i\}$ respectively.[9] We can break down this process in two steps. The first corresponds to changing the structure of the key-derivation graph. The leaf v_n needs to be removed from the tree rooted at v_I while a new leaf v'_n (Owned by party n) has to be added to the tree rooted at $v_{I'}$. If $I' = \emptyset$ no leaf is added. It is possible that the node $v_{I'}$, i.e, a node that has paths to exactly the nodes v_{S_j} for $j \in I'$, is not yet part of the lattice graph and has to be added as well. Note that in the case $I' = I \cup \{i\}$ the new node $v_{I'}$ can be connected in the lattice graph using the two edges $(v_{I'}, v_I)$ and $(v_{I'}, v_{\{i\}})$. If $I' = I \setminus \{i\}$ more edges might be necessary. After $\mathcal{G}_{\text{lat}}$ has been updated the Huffman trees with leaf $v_{I'}$ have to be updated.

As, after changing the structure of $\mathcal{G}$, the invariant that every party only knows the secret keys corresponding to the descendants of their leaf no longer holds, in a second step key material needs to be refreshed. More precisely all keys corresponding to descendants $\mathcal{D}(v_n)$ of n's former leaf have to be replaced with fresh ones, and similarly all key material corresponding to $\mathcal{D}(v'_n)$ has to be refreshed starting with leaf key $\mathrm{sk}_{v'_n}$ that has to be accessible to n. We discuss how this can be implemented for CGKA and Multicast in greater detail below.

[9] One could also imagine a more general operation $\mathrm{Change}(n, I')$ subsuming Add and Rem which changes n's index set to $I' \subseteq [k]$. The techniques of this section easily extend to this setting.

Continuous group-key agreement. In the setting of continuous group-key agreement there exists no central authority that holds all secret keys and administers structural changes in the groups. Accordingly, the action of adding party n to group S_i or removing n from S_i has to be initiated by a party m. To be able to do so without having party m sample key material for nodes outside of $\mathcal{D}(v_m)$, we will rely on the techniques of *blanking* paths and *unmerged leaves* of [BBR+23]. To every node v in the key-derivation graph $\mathcal{G}$ we associate a flag $\texttt{blanked} \in \{0,1\}$ and a list of unmerged leaves $\mathrm{Unm}(v)$. Finally the *resolution* $\mathrm{Res}(v)$ of v is defined as follows.

$$\begin{array}{ll} \mathrm{Res}(v) & = \{v\} \qquad\qquad \text{if } \texttt{blanked}(v) = 0 \\ \mathrm{Res}(v) & = \bigcup_{v' \in \mathcal{P}(v)} \mathrm{Res}(v') \qquad \text{else} \end{array}$$

where $\mathcal{P}(v)$ denotes the parents of v in $\mathcal{G}$. Intuitively, $\mathrm{Unm}(v)$ indicates leaves below v that have not yet been integrated in the key derivation graph, and $\mathrm{Res}(v)$ is used to address all leaves under v with a minimal set of unblanked nodes.

As discussed above, adding or removing user n from a group proceeds in two steps, the first computing structural changes in $\mathcal{G}$, the second refreshing key material. Let v_n denote the "old" leaf of n and v'_n the new leaf to be added to the Huffman tree with root $v_{I'}$.

To carry out the first step, firstly, v_n is removed from $\mathcal{G}$ and all remaining nodes v in $\mathcal{D}(v_n) \setminus \{v_n\}$ blanked, i.e. $\texttt{blanked}(v) \leftarrow 1$, the keys $(\mathrm{pk}_v, \mathrm{sk}_v)$ deleted, and the resolution $\mathrm{Res}(v)$ updated accordingly. Secondly, the new leaf v'_n is added to the Huffman tree rooted at $v_{I'}$, and for every $v \in \mathcal{D}(v'_n)$ the list of unmerged leaves is updated to $\mathrm{Unm}(v) \leftarrow \mathrm{Unm}(v) \cup \{v'_n\}$. Finally, if the operation was of the form $\mathrm{Add}(n,i)$ additionally the group key corresponding to i is deleted. Note that in the setting where the whole structure of $\mathcal{G}$ is known to the initiating party m, all changes can be computed by m and then communicated to the remaining parties via the delivery server. In a setting where users only know the part of $\mathcal{G}$ relevant to them, i.e., $\mathcal{D}(v_m)$ and the public keys of the co-parents with respect to $\mathcal{D}(v)$, m simply poses a request of the form $\mathrm{Add}(n,i)$ or $\mathrm{Rem}(n,i)$ to the server, which in turn computes the changes in $\mathcal{G}$ and sends personalized messages to all parties.

As to the second step, note that all group keys corresponding to $I \cup I'$ have been deleted. Thus, in order to resume communication in group $S_i \in I \cup I'$ a member m of this group has to perform an update - this is similar to the case of [BBR+23]. We now highlight how updates with respect to blanked nodes and unmerged leaves are computed compared to the version for a static key-derivation graph described in Section 4.3.2. Let $\mathcal{T}_m$ be the (in the case of our algorithm, unique) spanning tree of $\mathcal{D}(v)$. Then, starting from leaf v_m, new key pairs $(\mathrm{pk}_v, \mathrm{sk}_v)$ are generated from seeds Δ_v for all $v \in \mathcal{D}(v)$ in the way defined in Section 4.3.2. The set of ciphertexts corresponding to v is computed

starting from leaf v_m as follows. For every co-parent $v' \in \mathcal{CP}(v, \mathcal{T}_m)$ with respect to the spanning tree and every $v'' \in \mathrm{Res}(v') \cup \mathrm{Unm}(v')$, a ciphertext $\mathsf{Enc}(\mathrm{pk}_{v''}, \Delta_v)$ is generated. After computing the ciphertexts corresponding to v, the flag $\mathtt{blanked}(v)$ is set to 0, the resolution is recomputed accordingly, and the set of unmerged leaves is updated to $\mathrm{Unm}(v) \leftarrow \emptyset$. The update message consists of all ciphertexts.

Consider the case that an operation $\mathrm{Rem}(n, i)$ was carried out followed by an update by party $m \in S_i$. Since by correctness $v_{S_i} \in \mathcal{D}(v_m)$, the group key for S_i was refreshed. Further, all nodes with key material known to n were blanked (except for the new leaf v'_n, which does not have a path to v_{S_i}). This implies that all new keys generated by m were encrypted to keys not known by n, and so, n does not have access to the new group key for S_i. Note that all users in $S_i \setminus \{n\}$ have a path from their leaf to one of the v'' and hence are able to derive the new group key.

Now assume that a user $m \in S_j$ where j is an element of n's new index set I', performed an update. Then there must exist a node in $\mathcal{D}(v_m) \cap \mathcal{D}(v'_n)$. All elements of $\mathcal{D}(v'_n)$ contain v'_n as an unmerged leaf. Thus m must have encrypted a seed to $\mathrm{pk}_{v'_n}$ from which n can derive the new group key of S_j. A similar argument shows that the algorithm works as intended also for operation $\mathrm{Add}(n, i)$.

Summing up, if an add or remove operation for party n was carried out and I' and v'_n denote $n's$ new index set and leaf respectively, then:

(i) If a party $m \in S_j$ performs an update, then n can derive the group key of S_j exactly if $j \in I'$.

(ii) The graph invariant still holds, i.e., if n knows the secret key corresponding to node $v \in \mathcal{G}$ then it must hold that $v \in \mathcal{D}(v'_n)$.

Multicast. In the setting of multicast encryption, a central authority holds all keys ssk_v with $v \in \mathcal{G}$ and administers changes in the group structure. This makes adding users to or removing them from groups considerably easier. As discussed above, assume that the index set of party n changes from I to I' and let v_n and v'_n denote the old deleted leaf and the new leaf, respectively. To refresh the keys in $\mathcal{D}(v_n)$ the central authority derives them from a random seed and computes the corresponding ciphertexts as discussed in Section 4.3.2. Similarly, starting from a seed $\Delta_{v'_n}$ all key material for nodes in $\mathcal{D}(v'_n)$ is resampled and corresponding ciphertexts are prepared. Note that n needs to be given access to $\Delta_{v'_n}$. An easy way to is to simply update the old leaf seed by hashing it with a secure hash function, i.e, by setting $\Delta_{v'_n} \leftarrow \mathrm{H}'(\Delta_{v_n})$. Now n can compute the new seed locally.

4.7 Lower Bound on the Update Cost of CGKA

In this section we prove a lower bound on the average update cost of continuous group-key agreement schemes for multiple groups. As an intermediate step we will further prove a bound on the update cost of key-derivation graphs. To this aim, we follow the approach of Micciancio and Panjwani [MP04], who analyzed the worst-case communication complexity of multicast key distribution in a *symbolic* security model, where cryptographic primitives are considered as abstract data types. We will first recall their security model, adapt it to CGKA, and then prove how to extend their results to our setting. In Appendix 4.9, using a similar approach, we prove a lower bound for multicast encryption.

4.7.1 Symbolic Model

We first define a symbolic model in the style of Dolev and Yao [DY83] for CGKA schemes. It follows the approach of Micciancio and Panjwani [MP04], but as it admits the uses of public-key encryption also includes elements of the model of Bienstock et al. [BDR20], who analyze the communication cost of concurrent updates in CGKA schemes.

Building blocks. We restrict the analysis to schemes that are constructed from the following three primitives. Note that our construction is a special case of the constructions analyzed in this section.

- *Public-key Encryption:* Let $(\mathsf{KGen}, \mathsf{Enc}, \mathsf{Dec})$ denote a public-key encryption scheme, where
  - KGen on input of secret key sk returns the corresponding public key pk.
  - Enc takes as input a public key pk and a message m, and outputs a ciphertext $c \leftarrow \mathsf{Enc}(\mathrm{pk}, \mathrm{m})$.
  - Dec takes as input a secret key sk and a ciphertext c, and outputs a message $\mathrm{m} = \mathsf{Dec}(\mathrm{sk}, c)$. We assume perfect correctness: $\mathsf{Dec}(\mathrm{sk}, \mathsf{Enc}(\mathrm{pk}, \mathrm{m})) = \mathrm{m}$ for all sk, $\mathrm{pk} = \mathsf{KGen}(\mathrm{sk})$, and messages m.
- *Pseudorandom generator:* The algorithm G takes as input a secret key sk and expands it to a sequence of keys $\mathsf{G}_0(\mathrm{sk}), \ldots, \mathsf{G}_\ell(\mathrm{sk})$.
- *Secret sharing:* Let S, R denote the sharing and recovering procedures of a secret sharing scheme: For some access structure $\Gamma \subseteq 2^{[h]}$, the algorithm S takes as input a message m and outputs a set of shares $\mathsf{S}_1(\mathrm{m}), \ldots, \mathsf{S}_\mathsf{h}(\mathrm{m})$ such that for any $I \in \Gamma$ it holds $\mathsf{R}(\mathsf{I}, \{\mathsf{S}_\mathsf{i}(\mathrm{m})\}_{\mathsf{i}\in\mathsf{I}}) = \mathrm{m}$, but for any $I \not\subseteq \Gamma$ the message m cannot be recovered from $\{S_i(\mathrm{m})\}_{i\in I}$.

We consider the following data types that can be derived from other objects according to the following rules.

Data type		Grammar rules
Message m	$\leftarrow$	$\mathrm{sk}, \mathrm{pk}, \mathsf{Enc}(\mathrm{pk}, \mathrm{m}), S_1(\mathrm{m}), \ldots, S_h(\mathrm{m})$
Public key pk	$\leftarrow$	$\mathsf{KGen}(\mathrm{sk})$
Secret key sk	$\leftarrow$	$R, \mathsf{G}_0(\mathrm{sk}), \ldots, \mathsf{G}_\ell(\mathrm{sk})$

To describe the information that can be recovered from a set of messages M, the *entailment relation* is defined by the following rules:

$$\begin{aligned} \mathrm{m} \in M &\quad\Rightarrow\quad M \vdash \mathrm{m} \\ M \vdash \mathrm{sk} &\quad\Rightarrow\quad M \vdash \mathsf{G}_0(\mathrm{sk}), \ldots, \mathsf{G}_\mathsf{l}(\mathrm{sk}) \\ M \vdash \mathsf{Enc}(\mathrm{pk}, \mathrm{m}), \mathrm{sk} : \mathrm{pk} = \mathsf{KGen}(\mathrm{sk}) &\quad\Rightarrow\quad M \vdash \mathrm{m} \\ \exists I \in \Gamma : \forall i \in I : M \vdash \mathsf{S}_\mathsf{i}(\mathrm{m}) &\quad\Rightarrow\quad M \vdash \mathrm{m} \end{aligned}$$

By restricting to these relations we essentially assume *secure* encryption and secret sharing schemes. Examples and further comments (in the setting of multicast encryption) can be found in [MP04, Section 3.2]. The set of messages which can be recovered from M using relation $\vdash$ is denoted by $\mathsf{Rec}(\mathsf{M})$.

Continuous group-key agreement. We now define continuous group-key agreement protocols in the symbolic model. We consider the case of CGKA for a static system of users $[N]$ and groups $S_1, \ldots, S_k \subseteq [N]$. Note that a lower bound for schemes in this setting in particular also excludes schemes which allow dynamic operations, i.e., adding and removing users from groups.

A CGKA scheme for $[N]$ and $S_1, \ldots, S_k$ specifies two procedures:

- Initially, Setup assigns each user $n \in [N]$ a personal set SK_n^0 of secret keys. Furthermore, Setup generates a set $\mathrm{msgs}(0)$ of so-called *rekey* messages to establish for every group S_j a group secret key $\mathrm{sk}_{S_j}^0$. We require that the initial sets of personal keys consist of uniformly random keys, and that for all $n' \neq n$ and $\mathrm{sk} \in \mathrm{SK}_n^0$ we have $\mathrm{sk} \notin \mathsf{Rec}(\mathrm{SK}_{\mathsf{n}'}^0, \mathrm{msgs}(0))$.
- In round t, the algorithm Update takes as input a user identity $n \in [N]$, establishes new sets $\mathrm{SK}_{n'}^t$ for all users n', and outputs some rekey messages $\mathrm{msgs}(t)$ to establish for every group S_j an epoch t group key $\mathrm{sk}_{S_j}^t$. We do *not* require the new sets and group keys to be distinct from the ones of round $t-1$. We denote the set of new uniformly random keys that were generated during the update procedure by the updating party by F_n^t.

Note that the only party generating new keys during update t is the updating party n. For ease of notation we define $\mathrm{F}^t_{n'} = \emptyset$ for all $n' \neq n$, and set $\mathrm{F}^0_{n'} = \emptyset$ for all n'.

For *correctness*, we require that, (a) at all times members of a group are able to derive the current group key from their set of personal keys and the sent messages, and (b) if some user updated in round t, then all users are able to derive their new set of personal keys from their old one, the sent messages, and in the case of the updating party the new keys generated during the update. The latter condition accounts for the fact that changes to a user's set of personal keys need to be communicated to them.

More precisely, for (a) we require that for any subgroup structure and any sequence of updating users $(n_1, \ldots, n_t)$, for all $j \in [k]$ each member n of subgroup S_j can recover $\mathrm{sk}^t_{S_j}$:

$$\mathrm{sk}^t_{S_j} \in \mathsf{Rec}\Big(\mathrm{SK}^{\mathsf{t}}_{\mathsf{n}} \cup \bigcup_{\iota \in [\mathsf{t}]_0} \mathrm{msgs}(\iota)\Big) \ .$$

For (b) we require that for any subgroup structure and any sequence of updating users $(n_1, \ldots, n_t)$, we have for all n that

$$\mathrm{SK}^t_n \subseteq \mathsf{Rec}\Big(\mathrm{SK}^{\mathsf{t-1}}_{\mathsf{n}} \cup \mathrm{F}^{\mathsf{t}}_{\mathsf{n}} \cup \bigcup_{\iota \in [\mathsf{t}]_0} \mathrm{msgs}(\iota)\Big) \ .$$

For security, we assume the minimal requirement of *post-compromise security* (PCS), which essentially says that users can recover from compromise, which leaks their state and the keys generated during the time period of being compromised, by updating. Note that a lower bound in this setting in particular excludes protocols achieving stronger security notions desired in practice, like post compromise forward security [ACJM20].

More precisely, we formalize PCS as the condition that no group key can be recovered from members outside the group, and/or members' personal keys and the keys generated by them before their last update. To this end, for round t and user $n \in [N]$, let $t_{\mathrm{up}}(t, n)$ denote the round in which n performed their last update, where we set $t_{\mathrm{up}}(t, n) = 0$ if no such update occurred. I.e., we require that for any group system, any update pattern, in every round t we have that

$$\mathrm{sk}^t_{S_j} \notin \mathsf{Rec}\Big(\bigcup_{\substack{n \in [N] \setminus S_j, \\ t' \in [t]_0}} (\mathrm{SK}^{\mathsf{t}'}_{\mathsf{n}} \cup \mathrm{F}^{\mathsf{t}'}_{\mathsf{n}}) \cup \bigcup_{\mathsf{n} \in \mathsf{S_j}} \bigcup_{\mathsf{t}' \in [\mathsf{t}_{\mathrm{up}}(\mathsf{t},\mathsf{n})-1]_0} (\mathrm{SK}^{\mathsf{t}'}_{\mathsf{n}} \cup \mathrm{F}^{\mathsf{t}'}_{\mathsf{n}}) \cup \bigcup_{\mathsf{t}' \in [\mathsf{t}]_0} \mathrm{msgs}(\mathsf{t}')\Big) \ .$$

Note that in the definition above, excluding all sets of personal secret keys since a user's last update is necessary even in the case that another user's update might have

replaced them before round t, as otherwise SK_n^t and in turn $\mathrm{sk}_{S_j}^t$ could trivially be recovered by the two correctness conditions.

Our goal is to derive a lower bound on the communication complexity of CGKA schemes achieving PCS, i.e., the number of messages $|\cup_{t' \in [t]_0} \mathrm{msgs}(t')|$ sent by the protocol.

Key Graphs. The execution of any CGKA scheme can be reflected by a graph structure representing recoverability of the keys involved (cf. [MP04]). To define this graph, we first need to recall the definition of *useful* keys and messages.

A secret key sk is called *useless at time* t if it can be recovered from old key material, i.e., if

$$\mathrm{sk} \in \mathsf{Rec}\Big(\bigcup_{\mathsf{n} \in [\mathsf{N}]} \bigcup_{\mathsf{t}' \in [\mathsf{t}_{\mathrm{up}}(\mathsf{t},\mathsf{n})-1]_0} (\mathrm{SK}_{\mathsf{n}}^{\mathsf{t}'} \cup \mathrm{F}_{\mathsf{n}}^{\mathsf{t}'}) \cup \bigcup_{\mathsf{t}' \in [\mathsf{t}]_0} \mathrm{msgs}(\mathsf{t}') \Big) \ ,$$

otherwise sk is called *useful*. As we will show below, if a CGKA scheme satisfies correctness and post-compromise security, then for all $t \in \mathbb{N}$, $n \in [N]$, $j \in [k]$ it must hold that at least one of the user's personal keys $\mathrm{sk}_n^t \in \mathrm{SK}_n^t$ as well as all group keys $\mathrm{sk}_{S_j}^t$ are useful at time t.

To decide whether a message is useful, one has to consider the information it contains, where messages can be arbitrarily nested applications of encryption Enc and secret sharing S. Thus, a message m is said to *encapsulate* a (pseudo)random key sk if $\mathrm{m} = e_1(e_2(\ldots(e_j(\mathrm{sk}))\ldots))$ where $e_i = \mathsf{Enc}_{\mathrm{pk}_i}$ or $e_i = \mathsf{S}_{\mathsf{h}_i}$ (for some public key pk_i and $h_i \in [h]$). A message is then called *useful* if it encapsulates a useful key.

Defintion 4.7.1 (Key graph [MP04])**.** *The* key graph $\mathcal{KG}_t = (\mathcal{V}_t, \mathcal{E}_t)$ *for a CGKA scheme at time* t *is defined as follows.* $\mathcal{V}_t$ *consists of all the keys that are useful at time* t*, and* $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ *consists of all ordered pairs* $(\mathrm{sk}_1, \mathrm{sk}_2)$ *such that one of the following is true:*

1. *There exists* $j \in [l]$ *auch that* $\mathrm{sk}_2 = \mathsf{G}_{\mathsf{j}}(\mathrm{sk}_1)$*.*
2. *There exists a message* $\mathrm{m} \in \bigcup_{j \in [t]_0} \mathrm{msgs}(j)$ *with* $\mathrm{m} = e_1(\mathsf{Enc}(\mathrm{pk}_1, e_2(\mathrm{sk}_2)))$ *with* $\mathrm{pk}_1 = \mathsf{KGen}(\mathrm{sk}_1)$*. Here* e_1 *and* e_2 *are some sequences of encryption and secret sharing, and we require that* e_2 *does not contain any encryption under a public key that has a matching secret key that is useful at time* t*.*

Edges of the second type are called communication edges*.*

One can show that for any node sk in $\mathcal{KG}$ there is at most one edge of the first type incident to sk (the proof is analogous to [MP04, Proposition 1]). Note that edges of the first type do not incur any communication cost, while edges of the second type

require at least one message. Thus, in the following we will be interested in the number of communication edges. To this aim, we prove the following properties of key graphs. In particular, we show that even if a CGKA scheme does not rely on the use of a fixed key-derivation graph as discussed in Section 4.3, after every update the key graph must still have the properties of Definition 4.3.1.

We will rely on the following Lemma that can be proved analogously to [MP04, Lemma 1].

Lemma 4.7.2. *Consider a secure and correct CGKA scheme for $N \in \mathbb{N}$, $S_1, \ldots, S_k \subseteq [N]$. Then, for any $t \in \mathbb{N}$ and sequence of updates $(n_1, \ldots, n_t)$, the corresponding key graph $\mathcal{KG}_t$ satisfies the following. For every set of keys SK, and key sk_2 that is useful at time t, such that $\mathrm{sk}_2 \in \mathsf{Rec}\left(\mathrm{SK} \cup \bigcup_{t' \in [t]_0} \mathrm{msgs}(t')\right)$, there exists a useful $\mathrm{sk}_1 \in \mathrm{SK}$ such that there is a path from sk_1 to sk_2 in $\mathcal{KG}_t$ that only consists of keys sk with $\mathrm{sk} \in \mathsf{Rec}\left(\mathrm{SK} \cup \bigcup_{t' \in [t]_0} \mathrm{msgs}(t')\right)$.*

Note that the converse of Lemma 4.7.2 is not true, since, for example, a message $\mathsf{Enc}(\mathrm{pk}_1, \mathsf{S}_1(\mathrm{sk}_2))$ with useful keys $\mathrm{sk}_1, \mathrm{sk}_2$ and $\mathrm{pk}_1 = \mathsf{KGen}(\mathrm{sk}_1)$ incurs an edge $(\mathrm{sk}_1, \mathrm{sk}_2)$ while sk_2 can only be recovered from sk_1 if $\{1\} \in \Gamma$.

4.7.2 Lower Bound on the Average Update Cost.

The communication complexity of a CGKA scheme after t updates is given by $\left|\bigcup_{t' \in [t]_0} \mathrm{msgs}(t')\right|$. To measure the efficiency of the scheme we will consider the *amortized communication complexity*

$$\mathrm{Com}_{\mathrm{A}} \coloneqq \left| \bigcup_{t' \in [t]_0} \mathrm{msgs}(t') \right| / t \ .$$

We now are ready to compute a bound on the expectation of $\mathrm{Com}_{\mathrm{A}}$ in the scenario where, in every round, the updating party is chosen uniformly at random. In Appendix 4.9 we prove an analogous bound for multicast encryption that improves on [MP04, Theorem 1] in two aspects. It generalizes the bound to the setting of several, potentially overlapping groups, and further gives a bound on the *average* communication complexity of updates, as opposed to a worst case bound.

Theorem 4.7.3. *Consider a CGKA scheme CGKA for $N \in \mathbb{N}$, $S_1, \ldots, S_k \subseteq [N]$ that is secure in the symbolic model. Then the expected amortized average communication cost after t updates is bounded from below by*

$$\mathbb{E}[\mathrm{Com}_{\mathrm{A}}] \geq (1 - 1/t) \cdot \frac{1}{N} \sum_{\emptyset \neq I \subseteq [k]} |P_I| \cdot \log(|P_I|) \ .$$

and the asymptotic (in the number of update operations) update cost of the protocol is at least $\frac{1}{N}\sum_{\emptyset\neq I\subseteq[k]}|P_I|\cdot\log(|P_I|)$.

Proof. We prove the result by showing that the average communication complexity after the tth update has size at least $(t-1)\frac{1}{N}\sum_{\emptyset\neq I\subseteq[k]}|P_I|\cdot\log(|P_I|)$. To this end, we will show that with every update on average at least $\frac{1}{N}\sum_{\emptyset\neq I\subseteq[k]}|P_I|\cdot\log(|P_I|)$ useful messages become useless. We will rely on the following claim.

Claim 2. *There exists a CGKA scheme* CGKA' *that is secure in the symbolic model such that:*

1. *If* CGKA *and* CGKA' *are executed with respect to the same update pattern, then their communication costs coincide.*
2. *Consider a sequence of* t *updates. For every* $t'<t$ *there exist a subgraph* $\mathcal{H}'_{t'}$ *of the keygraph* $\mathcal{G}'_{t'}$ *of* CGKA' *at time* t'*, such that for every* $n\in[N]$ *there exists a set* $V_n^{t'}$ *of useful sources of* $\mathcal{H}'_{t'}$ *with* $V_n^{t'}\subseteq\bigcup_{t''\in[t']_0}(\mathrm{SK}_n^{t''}\cup\mathrm{F}_n^{t''})$ *such that* $(\mathcal{H}'_{t'},\{V_n^{t'}\})$ *satisfies the requirements of Lemma 4.4.1.*

Before proving the claim we show that it implies Theorem 4.7.3. To this end, recall, that at most one of the edges incident to a node in a key graph is not a communication edge. For $t'<t$ consider the key graph $\mathcal{G}'_{t'}$. By applying Lemma 4.4.1 to the subgraph $\mathcal{H}'_{t'}$ of $\mathcal{G}'_{t'}$ the number of useful messages encapsulating secret keys that can be reached from useful keys in $\bigcup_{t''\in[t']_0}(\mathrm{SK}_n^{t''}\cup\mathrm{F}_n^{t''})$ is on average at least $\frac{1}{N}\sum_{\emptyset\neq I\subseteq[k]}|P_I|\cdot\log(|P_I|)$.

Note that by definition of PCS all useful keys in $\bigcup_{t''\in[t']_0}(\mathrm{SK}_n^{t''}\cup\mathrm{F}_n^{t''})$ become useless if party n updates in the $(t'+1)$th round. By Lemma 4.7.2 all descendants of these keys and in turn messages encapsulating descendants become useless as well. We obtain that with update $(t'+1)$ on average at least $\frac{1}{N}\sum_{\emptyset\neq I\subseteq[k]}|P_I|\cdot\log(|P_I|)$ messages become useless. By linearity of expectation and since useless messages never become useful again this implies that after the t updates on average at least $(t-1)\frac{1}{N}\sum_{\emptyset\neq I\subseteq[k]}|P_I|\cdot\log(|P_I|)$ messages must have been sent in CGKA'. As CGKA by Claim 2 has the same communication cost as CGKA' this bound carries over to it. Now dividing by t yields the claim of the theorem.

All that remains to do is to prove Claim 2. We define CGKA′ to be the scheme that works uses the same initialization procedure as CGKA and computes updates in the same way, except that whenever a uniformly random secret key sk is generated in CGKA then CGKA' samples a uniformly random key sk' and sets $\mathrm{sk}\leftarrow\mathsf{G}_0(\mathrm{sk}')$ instead of the uniformly random key; this modified key sk is then used just the same as in CGKA in all further operations.

Note that the communication cost of both schemes coincides since CGKA' only makes additional calls to the pseudorandom generator but no additional use of the encryption and secret sharing schemes and that CGKA' preserves correctness. Further, CGKA' is secure since CGKA is secure: To see this, note that in the symbolic model there is no difference between a uniformly random key and a pseudorandom key, as long as the seed of the latter is not revealed. But the additional seeds sk' which we introduce in CGKA' are never used in any messages, nor are they used to derive any further keys; they only occur in the sets F_n^t where they replace the keys sk. Thus, security of CGKA' indeed directly follows from security of CGKA.

We now show that the second part of Claim 2 holds as well. In fact, for a sequence of t updates we will prove the following stronger statement. For all $t' < t$ (a) there exists a subgraph $\mathcal{H}'_{t'}$ of $\mathcal{G}'_{t'}$ with distinct nodes v_{S_i} and pairwise distinct sets $V_n^{t'} \subseteq \bigcup_{t'' \in [t']_0} (\mathrm{SK}_n^{t''} \cup \mathrm{F}_n^{t''})$ of sources such that

$$n \in S_i \quad \Rightarrow \quad \exists v_n \in V_n^{t'} \text{ such that there is a path from } v_n \text{ to } v_{S_i} \ ,$$

and that (b) for all $n \neq n'$ and $v \in V_n^{t'}$ it holds that $v \notin \mathsf{Rec}(\mathsf{V}_{\mathsf{n}'}^{\mathsf{t}'} \cup \mathrm{SK}_{\mathsf{n}'}^{\mathsf{t}'-1})$.

We argue inductively in t' that a subgraph and sets with the properties (a) and (b) must always exist. First consider the case $t' = 0$. Note that the group keys $v_{S_j} = \mathrm{sk}_{S_j}^0$ by definition are useful at time 0. Fix S_j and let $n \in S_j$. By correctness and Lemma 4.7.2 there exists a useful $v_{n,j} \in \mathrm{SK}_n^0$ that has a path to v_{S_j} in $\mathcal{G}'_0$. We define $V_n^0 = \{v_{n,j} \mid j : n \in S_j\}$ and $\mathcal{H}'^0$ to be the subgraph of $\mathcal{G}'^0$ induced by the union over n and j of paths from $v_{n,j}$ to v_{S_j}. Then the correctness condition of (a) holds and we only have to show that the V_n^0 consist of pairwise distinct sources. By definition we have $v_{n,j} \notin \mathsf{Rec}(\mathrm{SK}_{\mathsf{n}'}^0)$ for $n' \neq n$ implying that the V_n^0 are pairwise distinct, and further by Lemma 4.7.2 that $v_{n,j}$ in $\mathcal{H}'_0$ cannot be reached by any $v_{n',j'}$ with $n' \neq n$. Note that if $v_{n,j}$ can be reached by $v_{n,j'}$ with $j' \neq j$ then we can simply remove $v_{n,j}$ from V_n^0 without changing the correctness condition. Thus, we end up with pairwise disjoint sets V_n^0 of sources and (a) holds.

Now assume that (a) and (b) hold for all $t'' < t'$. Let $n_{t'}$ denote the party that issued update t'. First consider a party $n \neq n_t$ and a group S_j with $n \in S_j$. Note that by correctness and security the group key $v_{S_j} = \mathrm{sk}_{S_j}^t$ is useful at time t'. Further, by correctness we have

$$\mathrm{sk}_{S_j}^{t'} \in \mathsf{Rec}\Big(\mathrm{SK}_{\mathsf{n}}^{\mathsf{t}'} \cup \bigcup_{\mathsf{t}'' \in [\mathsf{t}']_0} \mathrm{msgs}(\mathsf{t}'')\Big) \quad \text{and} \quad \mathrm{SK}_{\mathsf{n}}^{\mathsf{t}'} \subseteq \mathsf{Rec}\Big((\mathrm{SK}_{\mathsf{n}}^{\mathsf{t}'-1} \cup \mathrm{F}_{\mathsf{n}}^{\mathsf{t}'}) \bigcup_{\mathsf{t}'' \in [\mathsf{t}']_0} \mathrm{msgs}(\mathsf{t}'')\Big) \ .$$

Thus, by Lemma 4.7.2 there exists a useful node $v \in \mathrm{SK}_n^{t'}$ with a path to v_{S_j} and a useful node $v' \in \mathrm{SK}_n^{t'-1}$ that has a path to v, where we used that $\mathrm{F}_n^{t'} = \emptyset$. As v' already existed at time $t' - 1$ and lies in $\bigcup_{t'' \in [t'-1]_0} (\mathrm{SK}_n^{t''} \cup \mathrm{F}_n^{t''})$ by the induction hypothesis

there must exist a node $v_{n,S_j} \in V_n^{t'-1}$ that is a source in $\mathcal{H}'_{t'-1} \subseteq \mathcal{G}'_{t'-1}$ and has a path to v' and in turn to v and v_{S_j}. We set $V_n^{t'} = \{v_{n,S_j} \mid j : n \in S_j\}$.

Now consider the party $n = n_{t'}$ that issued update t' and let S_j be such that $n \in S_j$. By the first correctness property we have $\mathrm{sk}_{S_j}^{t'} \in \mathsf{Rec}(\mathrm{SK}_{\mathsf{n}}^{\mathsf{t}'} \cup \bigcup_{\mathsf{t}' \in [\mathsf{t}']_0} \mathrm{msgs}(\mathsf{t}'))$. Since the node $v_{S_j} = \mathrm{sk}_{S_j}^{t'}$ is useful at time t' by Lemma 4.7.2 there exists a useful node $v \in \mathrm{SK}_n^{t'}$ with a path to v_{S_j}. Further, by the definition of PCS it is not possible that v can be recovered from $\mathrm{SK}_n^{t''}$ for any $t'' < t'$ and thus, must have been generated during the t'th update. More precisely, since n is the updating party, by security all elements of $\mathrm{SK}_n^{t'-1}$ are useless at time t' and since by correctness $v \in \mathsf{Rec}(\mathrm{SK}_{\mathsf{n}}^{\mathsf{t}'-1} \cup \mathrm{F}_{\mathsf{n}}^{\mathsf{t}'})$ we obtain by Lemma 4.7.2 that there exists useful $v_{n,j} \in \mathrm{F}_n^{t'}$ that has a path to v and in turn to v_{S_j}. Note that $v_{n,j}$ by definition of $\mathrm{F}_n^{t'}$ is a uniformly random secret key. By construction of CGKA' the only operation applied to $v_{n,j}$ was an application of G_0, which in particular implies that it never was encrypted under any key. Thus $v_{n,j}$ is a source in $\mathcal{G}'_{t'}$ and we can define $V_n^{t'} = \{v_{n,j} \mid j : n \in S_j\}$.

Now we can define $\mathcal{H}'_{t'}$ to be the subgraph of $\mathcal{G}'_{t'}$ induced by the union over n and j of paths from $v_{n,j}$ to v_{S_j}. Note that for any party $n \neq n_{t'}$ that did not update in round t' any $v_{n,j} \in V_n^{t'}$ can only be reachable from some other node $v \in (V_{n'}^{t'})$ with $n \neq n'$ in $\mathcal{H}_{t'}$ if during the t'th update it was encrypted under some key that can be recovered from $V_{n'}^{t'-1} \subseteq V_{n'}^{t'-1} \cup \mathrm{SK}_{n'}^{t'-1}$. This however, would contradict induction hypothesis (b). Thus all elements of $V_n^{t'}$ must indeed be sources in $\mathcal{H}_{t'}$.

Finally, note that (b) holds as well: For $n \neq n_t$ this follows from the induction hypothesis and correctness and for n_t as discussed above by construction of CGKA'.

□

4.8 Direct Comparison of Trivial Algorithm and Algorithm 1

Tighter Analysis of Example 4.5.1. We now give an analysis of Example 4.5.1 that shows that at least in such a situation our algorithm performs always better than the trivial solution, even including rounding. For simplicity, assume n_1 is a power of 2 (but k is arbitrary). Consider the following hypothetical construction of $\mathcal{T}_1$ and $\mathcal{T}_{1,2}$: first build the complete binary tree and keep all k users in $S_{12} = S_1 \cap S_2$ to the right. Let v be the node highest up in the tree such that all its parents are in S_{12}. Now remove all users in S_{12} from the tree and call the result $\mathcal{T}_1$. Build a new binary tree $\mathcal{T}_{1,2}$ (as balanced as possible) from the nodes in S_{12} and attach it to node v. Clearly, all users in $S_1 \setminus S_2$ have path length $\log n_1$ in $\mathcal{T}_1$ and node v has path length $\log n_1 - \lfloor \log k \rfloor$.

Also, all users in S_{12} have path length $< \lceil \log k \rceil < \lfloor \log k \rfloor + 1$ in $\mathcal{T}_{1,2}$. So we get

$$\mathbb{E}[\mathrm{len}(U_{\mathcal{T}_1})] = \frac{n_1 - k}{n_1} \log n_1 + \frac{k}{n_1}(\log n_1 - \lfloor \log k \rfloor) = \log n_1 - \frac{k}{n_1} \lfloor \log k \rfloor$$

and $\mathbb{E}[\mathrm{len}(U_{\mathcal{T}_{1,2}})] \leq \lfloor \log k \rfloor + 1$. Note that the same construction also works for $\mathcal{T}_2$ and yields

$$\mathbb{E}[\mathrm{len}(U_{\mathcal{T}_2})] = \log n_2 - \frac{k}{n_2} \lfloor \log k \rfloor$$

Since Huffman is optimal, creating $\mathcal{T}_1$, $\mathcal{T}_2$ and $\mathcal{T}_{1,2}$ by using Huffman cannot yield worse expected path lengths. Putting these together

$$\begin{aligned} \mathrm{Upd}(\mathcal{G}_{\mathrm{a1}}) &\leq n_1 \mathbb{E}[\mathrm{len}(U_{\mathcal{T}_1})] + n_2 \mathbb{E}[\mathrm{len}(U_{\mathcal{T}_2})] + k \mathbb{E}[\mathrm{len}(U_{\mathcal{T}_{1,2}})] \\ &\leq n_1 \log n_1 + n_2 \log n_2 - k(\lfloor \log k \rfloor - 1). \end{aligned}$$

Clearly, for $k \geq 2$ this is always negative.

Now we obtain an upper-bound in which the approach of the example and the use of Lemma 4.5.2 are combined in order to obtain a sufficient condition under which $\mathcal{G}_{\mathrm{a1}}$ outperforms $\mathcal{G}_{\mathrm{triv}}$. We generalize the example to build the trees that correspond to nodes of the form $v_{\{i\}}$ in $\mathcal{V}_{\mathrm{lat}}$ and then use Lemma 4.5.2 for the rest of nodes in $\mathcal{V}_{\mathrm{lat}}$.

Comparison of Trivial Algorithm and Algorithm 1. Let $S_1, \ldots, S_s \subseteq [N]$ and $\mathcal{G}_{\mathrm{lat}} = (\mathcal{V}_{\mathrm{lat}}, \mathcal{E}_{\mathrm{lat}})$ be the corresponding lattice graph. Let $T_{\{i\}} := \left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right| - \left|\mathrm{S}(v_{\{i\}})\right| = \sum_{\substack{J \subseteq 2^{[s]} \text{ st} \\ v_{\{i\} \cup J} \in \overline{\mathcal{P}}(v_{\{i\}}) \cap \mathcal{P}(v_J)}} \left|\mathrm{S}(\mathcal{A}(v_{\{i\} \cup J}))\right|$. For each $i \in [s]$ first build a binary tree (as balanced as possible) using all users in $\mathrm{S}(\mathcal{A}(v_{\{i\}}))$ and keeping $2^{\lfloor \log T_{\{i\}} \rfloor}$ users in $\bigcup_{v' \in \mathcal{P}(v_{\{i\}})} \mathrm{S}(\mathcal{A}(v'))$ to the right. Just as in the example, let v be the node highest up in the tree such that $\mathcal{P}(v) \subseteq \bigcup_{v' \in \mathcal{P}(v_{\{i\}})} \mathrm{S}(\mathcal{A}(v'))$. Then remove all users in $\bigcup_{v' \in \mathcal{P}(v_{\{i\}})} \mathrm{S}(\mathcal{A}(v'))$ from the tree and call the resulting tree $\mathcal{T}_{\{i\}}$. We build a Huffman tree, $\mathcal{T}^{aux}_{\{i\}}$, with $\left|\mathcal{P}(v_{\{i\}})\right|$ many leaves and weights $\mathrm{S}(\mathcal{A}(v'))$ for each $v' \in \left|\mathcal{P}(v_{\{i\}})\right|$ and attach it to v. Each leaf of $\mathcal{T}^{aux}_{\{i\}}$ corresponds to a node $v_{\{i\} \cup J}$ for some $J \subseteq 2^{[s]}$. We add an edge between each leaf of $\mathcal{T}^{aux}_{\{i\}}$ and the root of the corresponding $\mathcal{T}_{\{i\} \cup J}$. For $|I| \geq 2$ we just consider a Huffman tree $\mathcal{T}_I$. We can bound the update cost of $\mathcal{G}$ as $\mathrm{Upd}(\mathcal{G}) \leq \sum_{i \in [s]} (\mathrm{Upd}(\mathcal{T}_{\{i\}}) + \mathrm{Upd}(\mathcal{T}^{aux}_{\{i\}})) + \sum_{v_I \in \mathcal{V}_{\mathrm{lat}} \colon |I| \geq 2} \mathrm{Upd}(\mathcal{T}_I)$. We can upper-bound the update cost of $\mathcal{T}^{aux}_I$ using Lemma 4.5.2;

$$\mathrm{Upd}(\mathcal{T}^{aux}_{\{i\}}) \leq T_{\{i\}} \log T_{\{i\}} + T_{\{i\}} - \sum_{\substack{J \subseteq 2^{[s]} \text{ st} \\ v_{\{i\} \cup J} \in \overline{\mathcal{P}}(v_{\{i\}}) \cap \mathcal{P}(v_J)}} \left|\mathrm{S}(\mathcal{A}(v_{\{i\} \cup J}))\right| \log \left|\mathrm{S}(\mathcal{A}(v_{\{i\} \cup J}))\right|.$$

There exist $0 \leq a_{\{i\}}, b_{\{i\}} \leq 1$ such that $a_{\{i\}} + b_{\{i\}} = 1$, there are $a_{\{i\}} \left|\mathrm{S}(v_{\{i\}})\right|$ users that have path length at most $\lfloor \log \left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right| \rfloor$ in $\mathcal{T}_{\{i\}}$ and there are $b_{\{i\}} \left|\mathrm{S}(v_{\{i\}})\right|$ users

that have path length $\lceil\log\left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|\rceil$ in $\mathcal{T}_{\{i\}}$. The node v has path length at most $\lceil\log\left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|\rceil - \lfloor\log T_{\{i\}}\rfloor$ in $\mathcal{T}_{\{i\}}$. Therefore we have

$$\begin{aligned}
\mathrm{Upd}(\mathcal{T}_{\{i\}}) &\le \left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|\Bigg(\frac{a_I\left|\mathrm{S}(v_{\{i\}})\right|}{\left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|}\lfloor\log\left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|\rfloor + \frac{b_{\{i\}}\left|\mathrm{S}(v_{\{i\}})\right|}{\left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|}\lceil\log\left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|\rceil \\
&\quad + \frac{T_{\{i\}}}{\left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|}(\lceil\log\left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|\rceil - \lfloor\log T_{\{i\}}\rfloor)\Bigg) \\
&\le \left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|\Bigg(\frac{a_I\left|\mathrm{S}(v_{\{i\}})\right|}{\left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|}\lfloor\log\left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|\rfloor + \frac{b_{\{i\}}\left|\mathrm{S}(v_{\{i\}})\right|}{\left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|}\lceil\log\left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|\rceil \\
&\quad + \frac{T_{\{i\}}}{\left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|}(\lfloor\log\left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|\rfloor + 1 - \lfloor\log T_{\{i\}}\rfloor)\Bigg) \\
&\le \mathrm{Upd}(\mathcal{G}_{\text{triv}} \text{ of } S_i) + T_{\{i\}}(1 - \lfloor\log T_{\{i\}}\rfloor)
\end{aligned}$$

The last inequality follows from the fact that there cannot be less than $b_{\{i\}}\left|\mathrm{S}(v_{\{i\}})\right|$ users whose path length is $\lceil\log\left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right|\rceil$ in the tree constructed by the trivial algorithm.

We can upper-bound the update cost of $\mathcal{T}_I$ for $|I| \ge 2$ using Lemma 4.5.2;

$$\mathrm{Upd}(\mathcal{T}_I) \le |\mathrm{S}(\mathcal{A}(v_I))|(1 + \log|\mathrm{S}(\mathcal{A}(v_I))|) - \sum_{\substack{J\subseteq 2^{[s]} \text{ st} \\ v_{I\cup J}\in\overline{\mathcal{P}}(v_I)\cap\mathcal{P}(v_J)}} |\mathrm{S}(\mathcal{A}(v_{I\cup J}))|\log|\mathrm{S}(\mathcal{A}(v_{I\cup J}))|.$$

We sum over all $I \subseteq 2^{[s]}$ such that $v_I \in V'$ and we get

$$\begin{aligned}
\mathrm{Upd}(\mathcal{G}_{\mathrm{a1}}) &\le \sum_{i\in[s]}(\mathrm{Upd}(\mathcal{T}_{\{i\}}) + \mathrm{Upd}(\mathcal{T}^{aux}_{\{i\}})) + \sum_{v_I\in\mathcal{V}_{\text{lat}}:\,|I|\ge 2}\mathrm{Upd}(\mathcal{T}_I) \\
&\le \sum_{i\in[s]}\Bigg(\mathrm{Upd}(\mathcal{G}_{\text{triv}} \text{ of } S_i) + 3T_{\{i\}} - \sum_{\substack{J\subseteq 2^{[s]} \text{ st} \\ v_{\{i\}\cup J}\in\overline{\mathcal{P}}(v_{\{i\}})\cap\mathcal{P}(v_J)}} \left|\mathrm{S}(\mathcal{A}(v_{\{i\}\cup J}))\right|\log\left|\mathrm{S}(\mathcal{A}(v_{\{i\}\cup J}))\right|\Bigg) \\
&\quad + \sum_{v_I\in\mathcal{V}_{\text{lat}}:\,|I|\ge 2}\Bigg(|\mathrm{S}(\mathcal{A}(v_I))|(1 + \log|\mathrm{S}(\mathcal{A}(v_I))|) - \sum_{\substack{J\subseteq 2^{[s]} \text{ st} \\ v_{I\cup J}\in\overline{\mathcal{P}}(v_I)\cap\mathcal{P}(v_J)}} |\mathrm{S}(\mathcal{A}(v_{I\cup J}))|\log|\mathrm{S}(\mathcal{A}(v_{I\cup J}
\end{aligned}$$

Using the fact that $T_{\{i\}} \coloneqq \left|\mathrm{S}(\mathcal{A}(v_{\{i\}}))\right| - \left|\mathrm{S}(v_{\{i\}})\right| = \sum_{\substack{J\subseteq 2^{[s]} \text{ st} \\ v_{\{i\}\cup J}\in\overline{\mathcal{P}}(v_{\{i\}})\cap\mathcal{P}(v_J)}} \left|\mathrm{S}(\mathcal{A}(v_{\{i\}\cup J}))\right|$

yields

$$\mathrm{Upd}(\mathcal{G}_{\mathrm{a1}}) \leq \sum_{i\in[s]} \mathrm{Upd}(\mathcal{G}_{\mathrm{triv}} \text{ of } S_i) + \sum_{i\in[s]} \sum_{\substack{J\subseteq 2^{[s]} \text{ st} \\ v_{\{i\}\cup J}\in\overline{\mathcal{P}}(v_{\{i\}})\cap\mathcal{P}(v_J)}} \Big|\mathrm{S}(\mathcal{A}(v_{\{i\}\cup J}))\Big|(3-\log\Big|\mathrm{S}(\mathcal{A}(v_{\{i\}\cup J}))\Big|)$$
$$+ \sum_{v_I\in\mathcal{V}_{\mathrm{lat}}\colon |I|\geq 2} \left(|\mathrm{S}(\mathcal{A}(v_I))|(1+\log|\mathrm{S}(\mathcal{A}(v_I))|) - \sum_{\substack{J\subseteq 2^{[s]} \text{ st} \\ v_{I\cup J}\in\overline{\mathcal{P}}(v_I)\cap\mathcal{P}(v_J)}} |\mathrm{S}(\mathcal{A}(v_{I\cup J}))|\log|\mathrm{S}(\mathcal{A}(v_{I\cup J}))|\right)$$

For $|I|\geq 2$, the term $|\mathrm{S}(\mathcal{A}(v_I))|\log|\mathrm{S}(\mathcal{A}(v_I))|$ appears twice with a negative sign and once with a positive sign. Hence

$$\mathrm{Upd}(\mathcal{G}_{\mathrm{a1}}) \leq \mathrm{Upd}(\mathcal{G}_{\mathrm{triv}}) + \sum_{v_I\in\mathcal{V}_{\mathrm{lat}}\colon |I|=2} |\mathrm{S}(\mathcal{A}(v_I))|(7-\log|\mathrm{S}(\mathcal{A}(v_I))|)$$
$$+ \sum_{\substack{v_I\in\mathcal{V}_{\mathrm{lat}}\colon |I|>2 \\ \exists i\in[s]\colon v_I\in\mathcal{P}(v_{\{i\}})}} |\mathrm{S}(\mathcal{A}(v_I))|(4-\log|\mathrm{S}(\mathcal{A}(v_I))|) + \sum_{\substack{v_I\in\mathcal{V}_{\mathrm{lat}}\colon |I|>2 \\ \nexists i\in[s]\colon v_I\in\mathcal{P}(v_{\{i\}})}} |\mathrm{S}(\mathcal{A}(v_I))|(1-\log|\mathrm{S}(\mathcal{A}(v_I))|)$$

In particular, if

- $|\mathrm{S}(\mathcal{A}(v'))| \geq 2$ for every $v_I \in \mathcal{V}_{\mathrm{lat}}$ with $|I| > 2$ and such that $\nexists i \in [s]$ with $v_I \in \mathcal{P}(v_{\{i\}})$, and
- $|\mathrm{S}(\mathcal{A}(v'))| \geq 2^3 = 8$ for every $v_I \in \mathcal{V}_{\mathrm{lat}}$ with $|I| > 2$ and such that $\exists i \in [s]$ with $v_I \in \mathcal{P}(v_{\{i\}})$, and
- $|\mathrm{S}(\mathcal{A}(v'))| \geq 2^7 = 128$ for every $v_I \in \mathcal{V}_{\mathrm{lat}}$ with $|I| = 2$,

$\mathcal{G}_{\mathrm{a1}}$ outperforms $\mathcal{G}_{\mathrm{triv}}$. The first condition applies to nodes which do not correspond to the intersection of two of the original subsets. The second condition applies to nodes that correspond to the intersection of two subsets of $[N]$, of which exactly one is among the original subsets. The last condition applies to nodes with $|I| = 2$, that is, nodes that correspond to the intersection of S_i and S_j for some $i, j \in [s]$.

4.9 Multicast Encryption Lower Bound

In this Section we prove a lower bound on the average update cost of multicast encryption schemes for multiple groups. To this aim, we follow the approach of Micciancio and Panjwani [MP04], who analyzed the worst-case communication complexity of multicast key distribution in a *symbolic* security model, where cryptographic primitives are considered as abstract data types. We will first recall their security model and then prove how to extend their results to our setting.

4.9.1 Symbolic Model

We restrict the analysis to schemes that are constructed from the following three primitives. Note that our construction is a special case of the constructions analysed in this section.

- *Encryption:* Let (E, D) denote a symmetric-key encryption scheme, where
  - E takes as input a secret key ssk and a message m, and outputs a ciphertext $c \leftarrow \mathsf{E}_{\mathrm{ssk}}(\mathrm{m})$,
  - D takes as input a secret key ssk and a ciphertext c, and outputs a message $\mathrm{m} = \mathsf{D}_{\mathrm{ssk}}(\mathrm{c})$. We assume perfect correctness, i.e. $\mathsf{D}_{\mathrm{ssk}}(\mathsf{E}_{\mathrm{ssk}}(\mathrm{m})) = \mathrm{m}$ for all keys ssk and messages m. Furthermore, the encryption scheme is secure, i.e., informally, without knowledge of the key ssk one cannot recover m from c.
- *Pseudorandom generator:* The algorithm G takes as input a key ssk and expands it to a sequence of keys $\mathsf{G}_0(\mathrm{ssk}), \ldots, \mathsf{G}_\mathsf{l}(\mathrm{ssk})$, that are indistinguishable from a sequence $\mathrm{ssk}_0, \ldots, \mathrm{ssk}_l$ of uniformly random keys $\mathrm{ssk}_i \leftarrow R$ $(i \in [l]_0)$ without knowledge of the key ssk.
- *Secret sharing:* Let S, R denote the sharing and recovering procedures of a secret sharing scheme: For some access structure $\Gamma \subseteq 2^{[h]}$, the algorithm S takes as input a message m and outputs a set of shares $\mathsf{S}_1(\mathrm{m}), \ldots, \mathsf{S}_\mathsf{h}(\mathrm{m})$ such that for any $I \in \Gamma$ it holds $\mathsf{R}(\mathsf{I}, \{\mathsf{S}_\mathsf{i}(\mathrm{m})\}_{\mathsf{i}\in\mathsf{I}}) = \mathrm{m}$, but for any $I \not\subseteq \Gamma$ the message m cannot be recovered from $\{S_i(\mathrm{m})\}_{i\in I}$.

There are two types of data structures: messages and keys, which can be derived by repeatedly applying the above algorithms:

$$\mathrm{m} \leftarrow \{\mathrm{ssk},\, \mathsf{E}_{\mathrm{ssk}}(\mathrm{m}),\, \mathsf{S}_1(\mathrm{m}),\, \ldots,\, \mathsf{S}_\mathsf{h}(\mathrm{m})\}, \quad \mathrm{ssk} \leftarrow \{\mathsf{R},\, \mathsf{G}_0(\mathrm{ssk}),\, \ldots,\, \mathsf{G}_\mathsf{l}(\mathrm{ssk})\}$$

where R denotes some set of random keys. All functions $\mathsf{E}, \mathsf{G}_\mathsf{i}, \mathsf{S}_\mathsf{i}$ are assumed to output messages of approximately the same length as the keys in R; hence, the update cost can be measured as the number of transmitted messages.

To describe the information that can be recovered from a set of messages M, the *entailment relation* is defined by the following rules:

$$\begin{aligned}
\mathrm{m} \in M &\quad\Rightarrow\quad M \vdash \mathrm{m} \\
M \vdash \mathrm{ssk} &\quad\Rightarrow\quad M \vdash \mathsf{G}_0(\mathrm{ssk}), \ldots, \mathsf{G}_\mathsf{l}(\mathrm{ssk}) \\
M \vdash \mathsf{E}_{\mathrm{ssk}}(\mathrm{m}), \mathrm{ssk} &\quad\Rightarrow\quad M \vdash \mathrm{m} \\
\exists I \in \Gamma : \forall i \in I : M \vdash \mathsf{S}_\mathsf{i}(\mathrm{m}) &\quad\Rightarrow\quad M \vdash \mathrm{m}
\end{aligned}$$

By restricting to these relations we essentially assume *secure* encryption and secret sharing schemes. Examples and further comments can be found in [MP04, Section 3.2]. The set of messages which can be recovered from M using relation $\vdash$ is denoted by $\mathsf{Rec}(\mathsf{M})$.

A *multicast key distribution* protocol for a (static) set of N users and k subsets $S_1, \ldots, S_k \subseteq [N]$ consists of two components – the setup algorithm Setup and an update procedure Update. For simplicity, we assume that each user is member of at least one group.

- Initially, Setup assigns each user $n \in [N]$ a secret key $\mathrm{ssk}_{\{n\}}^{(0)}$, which is either a uniformly random key from R, or a pseudorandom key that was derived through a sequence of applications of G to another key ssk_n', i.e. $\mathrm{ssk}_{\{n\}}^{(0)} = \mathsf{G}_{\mathsf{l}_1}(\mathsf{G}_{\mathsf{l}_2}(\ldots(\mathsf{G}_{\mathsf{l}_j}(\mathrm{ssk}_{\mathsf{n}}'))\ldots))$ with $j \geq 0$, where ssk_n' must not coincide with any of the keys assigned to users in $[N]$. Furthermore, Setup generates a set $\mathrm{msgs}(0)$ of so-called *rekey* messages to establish group keys $\mathrm{ssk}_{S_i}^{(0)}$ (for all $i \in [k]$) among all members of the groups S_i.

- In round t, the algorithm Update takes as input a user identity $n \in [N]$, assigns this user a fresh key $\mathrm{ssk}_{\{n\}}^{(t)}$ and outputs some rekey messages $\mathrm{msgs}(t)$ to establish a fresh group key for each group of which i was a member. For all other members $n' \in [N] \setminus \{n\}$, we set $\mathrm{ssk}_{\{n'\}}^{(t)} := \mathrm{ssk}_{\{n'\}}^{(t-1)}$.

For *correctness*, we require that, for any adversarially chosen subgroup structure and any sequence of updating users $(n_1, \ldots, n_t)$, for all $j \in [k]$ each member i of subgroup S_j can recover $\mathrm{ssk}_{S_j}^{(t)}$, i.e.

$$\mathrm{ssk}_{S_j}^{(t)} \in \mathsf{Rec}\left(\{\mathrm{ssk}_{\{\mathsf{n}\}}^{(\mathsf{t})}\} \cup \bigcup_{\iota \in [\mathsf{t}]_0} \mathrm{msgs}(\iota)\right).$$

For security, we assume the minimal requirement of *post-compromise security*, namely that no group key can be recovered from members outside the group and/or old key material, i.e.

$$\mathrm{ssk}_{S_j}^{(t)} \notin \mathsf{Rec}\left(\bigcup_{\mathsf{n} \in [\mathsf{N}] \setminus \mathsf{S_j}} \{\mathrm{ssk}_{\{\mathsf{n}\}}^{(\mathsf{t})}\} \cup \bigcup_{\substack{n \in [N], \\ \iota \in [t-1]_0}} \left(\{\mathrm{ssk}_{\{\mathsf{n}\}}^{(\iota)}\} \setminus \{\mathrm{ssk}_{\{\mathsf{n}\}}^{(\mathsf{t})}\}\right) \cup \bigcup_{\iota \in [\mathsf{t}]_0} \mathrm{msgs}(\iota)\right).$$

4.9.2 Key Graphs

The execution of any multicast key distribution protocol can be reflected by a graph structure representing recoverability of the keys involved (cf. [MP04]). To define this graph, we first need to recall the definition of *useful* keys and messages.

A secret key ssk is called *useless at time* t if it can be recovered from old key material, i.e. if $\mathrm{ssk} \in \mathsf{Rec}\left(\bigcup_{\substack{n\in[N],\\ \iota\in[t-1]_0}} \left(\{\mathrm{ssk}^{(\iota)}_{\{\mathsf{n}\}}\} \setminus \{\mathrm{ssk}^{(\mathsf{t})}_{\{\mathsf{n}\}}\}\right) \cup \bigcup_{\iota\in[\mathsf{t}]_0} \mathrm{msgs}(\iota)\right)$, otherwise ssk is called *useful*. If a multicast key distribution protocol satisfies correctness and post-compromise security, then for all $t \in \mathbb{N}$, $n \in [N]$, $j \in [k]$ it must hold that the user's keys $\mathrm{ssk}^{(t)}_{\{n\}}$ as well as the group keys $\mathrm{ssk}^{(t)}_{S_j}$ are useful at time t.

To decide whether a message is useful, one has to consider the information it contains, where messages can be arbitrarily nested applications of encryption E and secret sharing S. Thus, a message m is said to *encapsulate* a (pseudo)random key ssk if $\mathrm{m} = e_1(e_2(\ldots(e_j(k))\ldots))$ where $e_i = \mathsf{E}_{\mathrm{ssk_i}}$ or $e_i = \mathsf{S}_{\mathsf{h_i}}$ (for some key ssk_i and $h_i \in [h]$). A message is then called *useful* if it encapsulates a useful key.

Defintion 4.9.1 (Key graph [MP04])**.** *The* key graph $\mathcal{KG}_t = (\mathcal{V}_t, \mathcal{E}_t)$ *for a multicast key distribution protocol at time* t *is defined as follows.* $\mathcal{V}_t$ *consists of all the keys that are useful at time* t*, and* $\mathcal{E} \subseteq \mathcal{V} \times \mathcal{V}$ *consists of all ordered pairs* $(\mathrm{ssk}_1, \mathrm{ssk}_2)$ *such that one of the following is true:*

- *There exists* $j \in [l]$ *auch that* $\mathrm{ssk}_2 = \mathsf{G_j}(\mathrm{ssk}_1)$.
- *There exists some message* $m \in \bigcup_{j\in[t]_0} \mathrm{msgs}(j)$ *such that* $m = e_1(\mathsf{E}_{\mathrm{ssk}_1}(\mathsf{e}_2(\mathrm{ssk}_2)))$ *for some sequences of encryption and secret sharing* e_1 *and* e_2*, and* e_2 *does not contain any encryption under a key that is useful at time* t.

Edges of the second type are called communication edges.

One can show that for any node ssk in $\mathcal{KG}$ there is at most one edge of the first type incident on ssk (see [MP04, Proposition 1] for a proof). Note that edges of the first type do not incur any communication cost; thus, in the following we will be interested the number of communication edges. To this aim, we prove the following properties of key graphs, which in particular show that key-derivation graphs as defined in Section 4.3.2 are just a special case of key graphs (cf. Definition 4.3.1).

Lemma 4.9.2. *Consider a secure multicast key distribution protocol for* $N \in \mathbb{N}$*,* $S_1, \ldots, S_k \subseteq [N]$*. Then, for any* $t \in \mathbb{N}$ *and sequence of updates* $(n_1, \ldots, n_t)$*, the corresponding key graph* $\mathcal{KG}_t$ *satisfies the following two conditions.*

1. *For* $n \in [N]$ *and* $j \in [k]$ *there exist nodes* v_n *and* v_{S_j} *in* $\mathcal{KG}_t$*, and for* $n \neq n' \in [N]$ *it holds* $v_n \neq v_{n'}$*.*

2. *For every pair of keys* $\mathrm{ssk}_1, \mathrm{ssk}_2$ *that are useful at time* t*, such that* $\mathrm{ssk}_2 \in \mathsf{Rec}\left(\{\mathrm{ssk}_1\} \cup \bigcup_{\iota \in [\mathsf{t}]_0} \mathrm{msgs}(\iota)\right)$*, there exists a path from* ssk_1 *to* ssk_2 *in* $\mathcal{KG}_t$ *that only consists of keys* ssk *such that* $\mathrm{ssk} \in \mathsf{Rec}\left(\{\mathrm{ssk}_1\} \cup \bigcup_{\iota \in [\mathsf{t}]_0} \mathrm{msgs}(\iota)\right)$*.*

Proof. By definition the keys $\mathrm{ssk}_{\{n\}}^{(t)}$ for users $n \in [N]$ are distinct and there exists a group key $\mathrm{ssk}_{S_j}^{(t)}$ for each group S_j ($j \in [k]$). To prove property 1, it remains to prove that these keys are useful, hence represent a node in $\mathcal{KG}_t$. For group keys $\mathrm{ssk}_{S_j}^{(t)}$ this follows immediately from security of the scheme. For the users' private keys, recall that we assume that for all $n \in [N]$ there exists some $j \in [k]$ such that $n \in S_j$. We assume for contradiction that the user's current key $\mathrm{ssk}_{\{n\}}^{(t)}$ was useless, i.e. could be recovered from users' old keys that have been replaced in rounds $[t]$ and the rekey messages. But by correctness it must hold that user n can recover the group key $\mathrm{ssk}_{S_j}^{(t)}$ from its own key and the rekey messages. This, however, would imply that $\mathrm{ssk}_{S_j}^{(t)}$ can be recovered from old keys and rekey messages, hence $\mathrm{ssk}_{S_j}^{(t)}$ would be useless – a contradiction.

For the second property, we refer to [MP04, Lemma 1] for a proof. □

Note that the converse of property 2 is not true, since e.g. a message $\mathsf{E}_{\mathrm{ssk}_1}(\mathsf{S}_1(\mathrm{ssk}_2))$ with useful keys $\mathrm{ssk}_1, \mathrm{ssk}_2$ incurs an edge $(\mathrm{ssk}_1, \mathrm{ssk}_2)$ while ssk_2 can only be recovered from ssk_1 if $\{1\} \in \Gamma$.

4.9.3 Lower Bound on the Average Update Cost

The communication complexity of a multicast encryption scheme after t updates is given by $\left|\bigcup_{\iota \in [t]_0} \mathrm{msgs}(\iota)\right|$. To measure the efficiency of the protocol we will consider the *amortized communication complexity*

$$\mathrm{Com}_{\mathrm{A}} := \left| \bigcup_{\iota \in [t]_0} \mathrm{msgs}(\iota) \right| / t \ .$$

We now are ready to compute a bound on the expectation of $\mathrm{Com}_{\mathrm{A}}$ in the scenario where in every round the updating party is chosen uniformly at random. The result improves on [MP04, Theorem 1] in two aspects. It generalizes the bound to the setting of several potentially overlapping groups, and further gives a bound on the *average* communication complexity of updates opposed to a worst case bound.

Theorem 4.9.3. *Consider a multicast key-distribution protocol for $N \in \mathbb{N}$, $S_1, \ldots, S_k \subseteq [N]$ that is secure in the symbolic model. Then the expected amortized average communication cost after t updates is bounded by*

$$\mathbb{E}[\mathrm{Com_A}] \geq (1 - 1/t) \cdot \frac{1}{N} \sum_{\emptyset \neq I \subseteq [k]} |P_I| \cdot \log(|P_I|) \ .$$

and the asymptotic update cost of the protocol is at least $\frac{1}{N} \sum_{\emptyset \neq I \subseteq [k]} |P_I| \cdot \log(|P_I|)$.

Proof. We prove the result by showing that the average communication complexity after the tth update has size at least $(t-1)\frac{1}{N} \sum_{\emptyset \neq I \subseteq [k]} |P_I| \cdot \log(|P_I|)$. To this end, we will show that with every update on average at least $\frac{1}{N} \sum_{\emptyset \neq I \subseteq [k]} |P_I| \cdot \log(|P_I|)$ useful messages become useless.

Let $1 \leq t' \leq t$. Consider the useful nodes v_n guaranteed to exist by Lemma 4.9.2, Property 1 after the $(t'-1)$st update (where the 0th update is to be understood as Setup). We show that the key graph $\mathcal{KG}_{t-1}$ contains a subgraph $\mathcal{G}'_{t-1}$ which satisfies the requirements of Lemma 4.4.1:

By Lemma 4.9.2, Property 2, for each $n \in [N]$ and $j \in [k]$ such that $n \in S_j$ there exists a path $\mathcal{P}_{n,j}$ from v_n to v_{S_j} in $\mathcal{KG}_{t-1}$ such that all keys associated to nodes in $\mathcal{P}_{n,j}$ can be recovered from $\mathrm{ssk}^{(t-1)}_{\{n\}}$ and the sent messages. Let $\mathcal{G}'_{t-1}$ denote the union of these paths. It remains to argue that all nodes v_n are sources in $\mathcal{G}'_{t-1}$. For contradiction, assume there exists $n, n' \in [N]$, $j \in [k]$ such that $n' \in \mathcal{P}_{n,j}$. But Update only replaces one user's private key; thus, if in the next round an update for n was generated, only n's private key would be replaced, but not n''s. Hence, security would be broken because n''s current key $\mathrm{ssk}^{(t)}_{\{n'\}} = \mathrm{ssk}^{(t-1)}_{\{n'\}}$ can be recovered from n's old key, and by correctness, for any $j' \in [k]$ such that $n' \in S_{j'}$, the key $\mathrm{ssk}^{(t)}_{S_{j'}}$ can be recovered from $\mathrm{ssk}^{(t)}_{\{n'\}}$. This proves that $\mathcal{G}'$ indeed satisfies the properties of Lemma 4.4.1.

Now, recall, that at most one of the edges incident to a node in the key graph is not a communication edge. Thus, by Lemma 4.4.1 the number of useful messages encapsulating keys that can be reached from v_n is on average at least $\frac{1}{N} \sum_{\emptyset \neq I \subseteq [k]} |P_I| \cdot \log(|P_I|)$.

Note that one of the keys v_n becomes useless after the t'th update. By Lemma 4.9.2, Property 2 all other nodes in $\mathcal{D}(v_n) \subseteq \mathcal{G}'_{t-1}$ and in turn messages encapsulating descendants become useless as well. With the argument above we obtain that with the t'th update on average at least $\frac{1}{N} \sum_{\emptyset \neq I \subseteq [k]} |P_I| \cdot \log(|P_I|)$ messages become useless. By linearity of expectation and since useless messages never become useful again this implies that after the t'th update on average at least $(t-1)\frac{1}{N} \sum_{\emptyset \neq I \subseteq [k]} |P_I| \cdot \log(|P_I|)$ messages have been sent. Now dividing by t yields the claim. □

4.10 Open problems

We conclude by discussing some open problems.

4.10.1 Optimal Key-derivation Graphs

Unfortunately we are not able to tell how far from optimal the solutions generated by Algorithm 1 are for concrete group systems. We consider it an interesting open question to resolve this issue.

General kdgs. We first discuss this problem in its general form. I.e., given a system $\mathcal{S} = \{S_1, \ldots, S_k\}$ of subgroups of the set $[N]$ of users compute the key-derivation graph for $\mathcal{S}$ (as defined in Definition 4.3.1) that has minimal update cost. The question of whether a polynomial time algorithm for solving this problem exists can be naturally asked in various ways. E.g., when polynomial means polynomial in the number of users N (think of N being given in unary), or polynomial in a reasonable description of the set system $\mathcal{S}$, say, when we are given the sizes of all non-empty intersections of sets in $\mathcal{S}$. Here N can be exponential in the input length, so a potential solution would need to have a very succinct description. Algorithm 1 (which we do not know whether is optimal) can be turned into one of the latter kind by using an implicit representation during the Huffman coding step.

We are thankful to one reviewer of this work, who pointed out an interesting connection of key-derivation graphs for a group system $\mathcal{S} = \{S_1, \ldots, S_k\}$ to the *disjunctive complexity* of $\mathcal{S}$, which, given variables $x_1, \ldots, x_N \in \{0, 1\}$, corresponds to the size of the smallest circuit of fanin-2 OR-gates computing

$$\bigvee_{i \in S_1} x_i, \ldots, \bigvee_{i \in S_k} x_i \ . \tag{4.12}$$

Note that circuits computing (4.12) correspond exactly to key-derivation graphs for $\mathcal{S}$. So the two problems differ only by the used metric; while disjunctive complexity counts the number of non-sources in the graph, the update cost of a kdg weighs each of these nodes by the number of sources below it. As there exist upper and lower bound on the disjunctive complexity of group systems (see e.g. [Juk12]), we consider it an interesting open questions whether these can be used to establish bounds on the update cost of kdgs. We want to point out, however, that this metric might be not fine-grained enough to capture certain properties of kdgs: E.g., for $N \in \mathbb{N}$ the systems $\mathcal{S}_1 = \{[N]\}$ and $\mathcal{S}_2 = \{[1], [2], \ldots, [N]\}$ both have disjunctive complexity $N - 1$, but their total update costs as kdgs are of order $N \cdot \log(N)$ and N^2 respectively.

Lattice based kdgs. If we restrict our view to algorithms using Boolean-lattice based graphs as defined in Section 4.5.4, and are willing to make simplifying assumptions,

the question of optimality translates to an optimization problem on graphs: we are going to (a) consider only lattice graphs $\mathcal{G}_{\mathrm{lat}}$ where all nodes v are connected with their descendants $v' \in \mathcal{D}(v)$ by an *unique* path, and (b) assume in our analysis of the update cost that the algorithms second step (i.e., the generation of Huffman trees) is instead implemented with an idealized code, that has average codeword length matching the entropy of the leaf distribution. This essentially corresponds to ignoring the terms of $+1$ in Lemma 4.2.1.

Recall that for groups system $\{S_1, \ldots, S_k\}$ the nodes $v_I \in \mathcal{V}_{\mathrm{lat}}$ of a lattice graph correspond to index sets $I \subseteq [k]$. It is easy to see that the correctness of $\mathcal{G}_{\mathrm{lat}}$ together with condition (a), are equivalent to requiring that the only sinks in the graph are the singleton sets $\{i\}$, and that for every $v_I \in \mathcal{V}_{\mathrm{lat}}$

$$I = I_1 \cup \cdots \cup I_\ell \tag{4.13}$$

holds, where $v_{I_1}, \ldots, v_{I_\ell}$ are the children of v_I and disjointness enforces unique paths.

The total update cost of a graph satisfying this property can be computed as follows. To every node v_I we associate the weight $w_I = \left|\bigcap_{i \in I} S_i \setminus \bigcup_{j \in [k] \setminus I} S_j\right|$ corresponding to the number of users exactly in the groups specified by I. Further, we inductively define the total weight t_I of v_I as

$$t_I = \begin{cases} w_I & \text{if } v_I \text{ is source} \\ w_I + \sum_{I' \colon v_{I'} \in \mathcal{P}(v_I)} t_{I'} & \text{else} \end{cases} ,$$

where $\mathcal{P}(v_I)$ denotes the set of parents of v_I. By assumptions (a) and (b), and Lemma 4.5.2, the update cost contributed by node v_I thus corresponds to

$$\mathrm{Upd}(v_I) = t_I \log(t_I) - \sum_{I' \colon v_{I'} \in \mathcal{P}(v_I)} t_{I'} \log(t_{I'}) \ , \tag{4.14}$$

and we end up with the following optimization problem on lattice graphs.

Problem 4.10.1. *Let $k \in \mathbb{N}$. Given weights $\{w_I\}_{I \subseteq [k]}$ with $w_I \in \mathbb{N}$ among the subgraphs of the Boolean lattice with respect to the power set of $[k]$ that satisfy Condition 4.13 find the subgraph $\mathcal{G}_{\mathrm{lat}}$ of minimal total update cost*

$$\mathrm{Upd}(\mathcal{G}_{\mathrm{lat}}) = \sum_{I \subseteq [k]} \mathrm{Upd}(v_I) \ .$$

We consider it an interesting open question whether Algorithm 1 solves this problem and, if not, to find an efficient algorithm that does.

4.10.2 Security

In this work we focused on the communication complexity of key-derivation graphs and only gave an intuition on their security. Security proofs for secure group messaging are typically quite complex, and protocols rely on additional mechanisms (e.g. confirmation tag, transcript hash, and parent hash) ensuring that users of the system can not be tricked into inconsistent views of the graph. We consider it an important open question to adapt these mechanisms to kdgs for several groups and give a formal security proof for the resulting CGKA protocols.

4.10.3 Efficiency of Dynamic Operations

As discussed in Section 4.6 the techniques of blanking and unmerged leaves can be adapted to key-derivation graphs in order to allow dynamic changes to the group membership. As is the case for singular groups, blanking and unmerged leaves decrease the efficiency of updates of a user n, since they destroy the binary structure of the graph, resulting in potentially more than a single ciphertext per node in $\mathcal{D}(v_n)$ having to be generated. However, the graph gradually recovers from this, assuming that parties with update trees overlapping $\mathcal{D}(v_n)$ update. It is an interesting open question how the decrease in efficiency compares to that of the trivial algorithm.

CHAPTER 5

CoCoA

5.1 Introduction

As mentioned in the introduction[1], most CGKA protocols today were designed with an asynchronous communication setting in mind, meaning that parties should be able to come online whenever and be able to receive and send messages and perform any group operations (add or remove members, or update their key material) regardless of the online status of the remaining group members.

The Problem Of Coordination. One property the first generation of CGKA protocols [KPPW+21, ACDT20, BBR18, CCG+18] share is that they require *all* protocol packets to be processed in exactly the same order by every group member. However, ensuring this level of coordination can present real challenges in a variety of settings; especially for large groups (e.g. with $50,000$ members as is targeted by the IETF's upcoming E2E secure messaging standard MLS [BBR+23]). In particular, it might lead to the problem sometimes called "starvation" where a client's packets are constantly rejected by the group (e.g. when the client is on a slow network connection and so can never distribute its own packets fast enough).

There do not seem to be any practical solutions to convincingly provide this level of coordination without significant drawbacks. Implementing the buffering mechanism via a single server does not automatically address the issue of starvation of clients with a slow connection. Nor is a round-robin "speaking slot" approach a satisfactory solution (even assuming universal time), as it would severely impact responsiveness; especially

[1] This Chapter essentially replicates, with permission, large parts of the full version [AAN+22c] of our publication [AAN+22b].

for larger groups. It is also not just responsiveness that suffers from a reduction of the rate at which parties can send new packets to the group. The quality of the security of a session (e.g. the speed with which privacy is recovered after a group member's local state is leaked) is also tightly dependent on the rate at which participants can send out packets. After all, if a compromised party has not even been able to send anything new to the group since a compromise, they cannot have updated the leaked cryptographic material to something the adversary cannot simply derive itself.

Concurrency At A High Price. To mitigate this problem, version 8 of MLS introduced a new syntax referred to as the "propose-and-commit" (P&C) paradigm. This has been adopted by most second generation CGKA protocols [ACJM20, AJM22] as it allows for some degree of concurrency. In particular, group members (and even designated external parties) may concurrently propose changes to the group state e.g. "Alice proposes adding Bob", "Charlie proposes updating his keys", etc. At any point, a group member can collect such proposal into a commit message which is broadcast to the group and actually effects all changes in the referenced proposals. Note, however, that commit messages must still be processed in a globally unique order. Moreover, in each of these protocols there is a high price being paid for large amounts of concurrency. Namely, the greater the number of proposals in a single commit message, the less efficient (e.g. greater packet size) certain future commits will be. In fact, efficiency can degenerate to the point where (starting from an arbitrary group state) a commit to $\Theta(n)$ proposals can produce a state where the next commit packet is forced to have size $\Omega(n)$; a far cry from the desired $O(\log(n))$.

Lower-Bounds on Communication Complexity. Bienstock et al. [BDR20] showed that there are limits to what we could hope for in terms of reducing communication complexity. Specifically, they show that T group members updating concurrently incurs a communication cost per user in the following round that is linear in T in any "reasonable" protocol.[2] In fact, if all n parties wish to update concurrently within 2 rounds then this has complexity at least $\Omega(n^2)$.

5.1.1 Our Contributions

In this paper we propose a new CGKA protocol called CoCoA (for *COncurrent COntinuous group key Agreement*) which is designed specifically to allow for efficient concurrent group operations. The way in which CoCoA handles *conflicts* of concurrent operations is very similar to the way the original TreeKEM paper [BBR18] suggested. The PCS

[2]For the lower bound, [BDR20] considered a symbolic model of execution which only applies to protocols constructed through black-box use of (possibly dual) PRFs, (possibly updatable) PKE, and broadcast encryption. Both our protocol and most TreeKEM variants fall into this category.

guarantees of such an approach were discussed since its inception[3], but saw no formal analysis. This approach was later dropped in future versions, perhaps due to the, at the time, unclear effect on PCS[4]. In this work, we formalize this approach into a protocol, and carefully analize the security guarantees it provides. Indeed, we show that in contrast to past CGKA protocols, healing may require more than 2 rounds (in the worst case $\log(n)$ rounds). However, even when all n users update their keys concurrently in $\log(n)$ rounds, the total communication complexity of any user is only roughly $(\log(n))^2$ (constant size) ciphertexts. This circumvents [BDR20] as their lower-bound only holds for updates that complete healing in at most 2 rounds. So, for the price of more interaction CoCoA can *greatly* decrease the actual bandwidth consumed.

To emphasize this even more, consider the cost of transitioning from a fully blanked tree to a fully unblanked one. We believe this to be a particularly interesting case as it captures the transition from any freshly created group into a bandwidth-optimal one. The faster/cheaper this transition can be completed, the faster an execution can begin optimal complexity behaviour. TreeKEM [BBR18], the CGKA scheme used in the MLS messaging protocol, needs $n/2$ rounds with receiver complexity, i.e. number of ciphertexts downloaded per user, $\Omega(n \log(n))$. The protocol in [BDR20], in turn, would be able to unblank the whole tree in 2 rounds with linear sender and recipient communication per user. In contrast, in CoCoA the tree could be unblanked in 1 round with linear sender cost, but only logarithmic recipient cost. For big groups this difference is very significant.

With such low communication, a user cannot learn all the $2n - 1$ fresh public-keys in the ratchet tree. Fortunately, users in CoCoA only need to know the $\log(n)$ secret keys and another $2\log(n)$ public keys. So in our protocol, users will not have a complete view of the public state as in previous protocols, but only know the partial state that is relevant to them. As a consequence, the server no longer acts as a relay but instead computes packets tailored to the individual receiving user. This comes with a new challenge that we address in this work: ensuring consistency across all users is not as straightforward anymore. This is crucial for security, since users disagreeing e.g. on the set of group members can lead to severe attacks.

Once we take into account operations like adding and removing group members, efficiency might degrade (though not to anything worse than past protocols). Nevertheless, in a typical execution we can expect to see far more updates than adds/removes. In particular, the more updates parties perform the faster the protocol heals from past compromises so it is generally in users' interest to perform updates as regularly as they can. By (greatly) reducing the cost of updates compared to past CGKA proto-

[3] https://mailarchive.ietf.org/arch/msg/mls/9u6BGEqWTfjDjbWaSmU2Z2JJniY/

[4] https://mailarchive.ietf.org/arch/msg/mls/HbFcflhaxfobZugCGI-jDLPDvGM/

cols, we allow groups to have quantitatively better security for the same amount of communication complexity spent.

In terms of security, we prove CoCoA secure in a "partially active setting", where, recall, the adversary can (repeatedly) leak parties local states including any random coins they use and query users to generate protocol messages. As opposed to the model of [KPPW+21], our adversary not only controls the delivery server but is also allowed to send arbitrary (potentially malformed) messages. Still, corruptions do not leak signature keys, thus considering insider attacks outside of our model. While the latter is a strong assumption, it is common in the literature, and we discuss it in more detail in Section 2.4.

5.1.2 Related Work

On top of the related work discussed in Section 1.3, we discuss here some works that are of particular relevance for this chapter.

Several works have studied CGKA's supporting varying degrees of concurrent operations, since it was first mentioned in TreeKEM's original proposal [BBR18]. In the first place, Weidner's Causal TreeKEM [Mat19] explores the idea of updates *re-randomizing* key material instead of overwriting it. Even though the protocol lacks forward secrecy and a complete security proof, it is the first to explicitly propose a protocol where updates do not need to come in a total order. This was followed by [WKHB21a], which proposed a decentralized CGKA protocol, thus admitting any level of un-coordination between the parties; albeit with linear communication complexity. Finally, a paper by Bienstock *et al.* [BDR20] studies the trade-off between PCS, concurrency and communication complexity, showing a lower bound for the latter and proposing a close to optimal protocol in their synchronous model for a fixed group in a weak security model (see Section 5.4).

Following the publication of [AAN+22b], which this Chapter is based on, the work of Alwen *et al.* [AAN+22a] proposed a protocol that achieves PCS concurrently in a number of rounds that only depends logarithmically in the number of corruptions. In particular, independent of the group size. Chapter 6 is based on this work.

The notion of *server-aided CGKA* was first formalized in [AHKM22]. They introduced the SAIK protocol, which reduces receiver communication costs by as much as 1000 fold in groups of size $10,000$. However, the earlier work of [DDF21] and the concurrent work of [HKP+21] both include (implicit) server-aided CGKAs as well.

Our protocol CoCoA is shown to be close to optimal in [ACNPPP23], which shows a lower bound on the communication complexity of healing in $\log(n)$ rounds which is a

factor of $\log(n)\log(\log(n))$ away from CoCoA's cost. This work further generalizes CoCoA to trees of arbitrary in-degree, which allows for healing in an arbitrary number k of rounds, and further proposed a modification of CoCoA that can achieve PCS in k rounds with a cost that is lower by a factor of roughly $k\sqrt[k/2]{n}$ (note that this is only an improvement for larger values of k).

5.2 Preliminaries

In this Chapter, we will use the definition of CGKA introduced earlier (Definition 2.3.1).

5.2.1 Ratchet Trees

Our protocol builds on TreeKEM, and thus uses the same underlying structure of a *ratchet tree* for deriving shared secrets among the group members, which was described in Section 2.5. Recall that a ratchet tree is a directed binary tree $\mathcal{T} = (V_{\mathcal{T}}, E_{\mathcal{T}})$, with edges pointing towards the root node $v_{root}$and each user in the group associated to a leaf. We will use the notation $\mathcal{T}^i = (V^i_{\mathcal{T}}, E^i_{\mathcal{T}})$ to refer to the ratchet tree associated to round i.

Node states. On top of the basic node state defined in Section 2.5, the state of nodes in CoCoA additionally contains the following information. A vector of public keys $\mathrm{PK}^{\mathrm{pr}}$ called the *predecessor keys* which correspond to the public keys of the nodes in the resolution of v in the round right before the current key pk_v was first introduced, see Section 5.3.4; a pair of hash values h_v called the parent hash of v; an identifier corresponding to the party ID_v generating the node's key pair (note that this is the case for *all* nodes); a signature σ_v under the private signing key of ID_v; a transcript hash value, $\mathcal{H}_{trans}$, committing to the state of ID_v at the time of sampling that node's key pair (defined below in Section 5.3.3); a confirmation tag value $\mathrm{confTag}$ (defined below in Section 5.3.4); an optional pair of hash values $o_v = (o_{v,1}, o_{v,2})$ corresponding to partial openings of a Merkle commitment sent by the server and encoding the state of the parent nodes of v; and a set of of so called *unmerged leaves* $\gamma(v).\mathrm{Unmerged}$, or simply $\mathrm{Unmerged}(v)$, corresponding to the leaves (and their associated public keys) of the subtree rooted at v whose users have no knowledge of sk_v (this will be the case, temporarily, for newly added users). In a slight abuse of notation, given a set of nodes S, we define its set of unmerged nodes to be $\mathrm{Unmerged}(S) = \cup_{v\in S}\mathrm{Unmerged}(v)$. Finally, for an internal node v, we will write ssk_v to refer to the secret signing key of party ID_v. The *secret part* of $\gamma(v)$ consists just of sk_v, and ssk_v in case v is a leaf. Looking ahead, parties might end up (through a misbehaving delivery server) having different views on the state of a given node, and so we will refer to the view of party ID_i of v at round n as $\gamma^n_i(v)$. For a summary of node states see Table 5.1.

$(\gamma(v).\mathrm{sk} = \mathrm{sk}_v, \gamma(v).\mathrm{pk} = \mathrm{pk}_v)$	The node's key-pair
$(\mathrm{ssk}_v, \mathrm{svk}_v)$	Signing key-pair (Only present if v is a leaf)
$\mathrm{PK}^{\mathrm{pr}}$	vector of predecessor keys
$h_v = (h_{v,1}, h_{v,2})$	Parent hash value
ID_v	Identifier of party setting v's key
σ_v	Signature under ID_v's key
$\mathcal{H}_{trans,v}$	Transcript hash
$\mathrm{confTag}_v$	Confirmation tag
$o_v = (o_{v,1}, o_{v,2})$	Pair of partial Merkle commitment openings
$\gamma(v).\mathrm{Unmerged} = \mathrm{Unmerged}(v)$	Set of unmerged leaves keys

Table 5.1: State $\gamma(v)$ of non-blank node v.

5.3 The CoCoA Protocol

We start with a high level description of the CoCoA protocol in Section 5.3.1. Section 5.3.2 covers users' states and the key schedule, Section 5.3.3 robustness and the round hash, Section 5.3.4 the parent hash mechanism, and Section 5.3.5 formally defines the protocol procedures.

5.3.1 Overview

Concurrent updates in CGKA. To recover from compromise, CGKA protocols allow users to refresh the secret key material known to them. Recall that a user does this by re-sampling all keys they know (those on the user's path in the case of a ratchet tree), encoding them in an update message, and sending this to the server, which broadcasts it to the other group members. However, it is unclear how to handle concurrent update attempts by several users.

As a first approach, it seems natural to simply reject all but one update. Using a fixed rule to determine whose update to implement, however, might lead to starvation, with users blocked from updating and thus not recovering from compromise (compare Fig. 5.1, column (a)). Even if parties that did not update for the longest time are prioritized, it may take a linear number of update attempts to fully recover security of the ratchet tree (compare Fig. 5.1, column (b)).

The more recent versions of the MLS protocol partially deal with this through the "propose and commit" paradigm (see Section 2.6). Roughly, update proposals refresh a user's leaf key and signal the intent to perform an update. A commit then allows a user to implement several concurrent update proposals. While this allows the ratchet tree to fully recover within two rounds, this comes at the cost of destroying the binary

structure of the tree, as, in order to preserve the tree invariant, nodes not on the path of the committing party are blanked. In the worst case, this can lead to future updates having a size linear in the number of parties (compare Fig. 5.1, column (c)).

The approach we take with the CoCoA protocol is to introduce concurrency as originally suggested in the pre-P&C versions, and implement all updates simultaneously, albeit some of them only partially. Intuitively, while the ratchet tree might not fully recover immediately, every updating party still makes progress towards recovery; and after logarithmically many updates of every compromised user, security is restored (compare Fig. 5.1, column (d)).

Updates in the CoCoA protocol. The main idea in the CoCoA protocol is, given several concurrent update messages, to apply all of them simultaneously, while resolving conflicts by means of an ordering of the operations. As a consequence, some updates might only be applied partially. More precisely, the protocol parameters contain an ordering $\prec$. This could be, e.g. the lexicographic ordering, however, the particular choice does not affect our security results. Then, given a set of update messages $\{U_1, \ldots, U_k\}$, if a node in the ratchet tree would be affected by several U_i, the one that is minimal with respect to $\prec$ takes precedence and replaces its key pair. Consider the example of Fig. 5.2, in which the users A, C, G in a group of size 8 concurrently update, with C's update taking precedence over the other two. Note that since the updates are concurrent, new keys get encrypted to keys of the previous round. Assume, e.g., that C and G were compromised. Then, after the updates, all compromised keys are replaced. However, only the first three keys in C's and G's update paths are secure, while the new Δ_{root} was encrypted to an old, compromised key and hence is known to the adversary. So, while the ratchet tree did not fully recover, it made progress towards it. In Section 5.5 we discuss the security of CoCoA in more detail.

Adds and Removes. To be able to add users to and remove users from the group, CoCoA combines features from different versions of TreeKEM. In general, it follows the approach of TreeKEM v7 in that it does not distinguish between proposals and commits. However, add operations are handled in a way reminiscent of the latter technique, as they are executed in two rounds: a first round where they get announced to the rest of group members, and a second where the parties actually join the group, after receiving a *welcome message*. We stress that add operations taking two rounds seems to be an inherent consequence of allowing concurrency: an updating user cannot compute encryptions for a user added to the group in the same round by a different party. Moreover, adds are executed following the *unmerged leaves* technique (see Section 2.6), which can be thought of as initially connecting a new user's leaf directly to v_{root} and progressively connecting it to lower nodes as the keys for these get rotated.

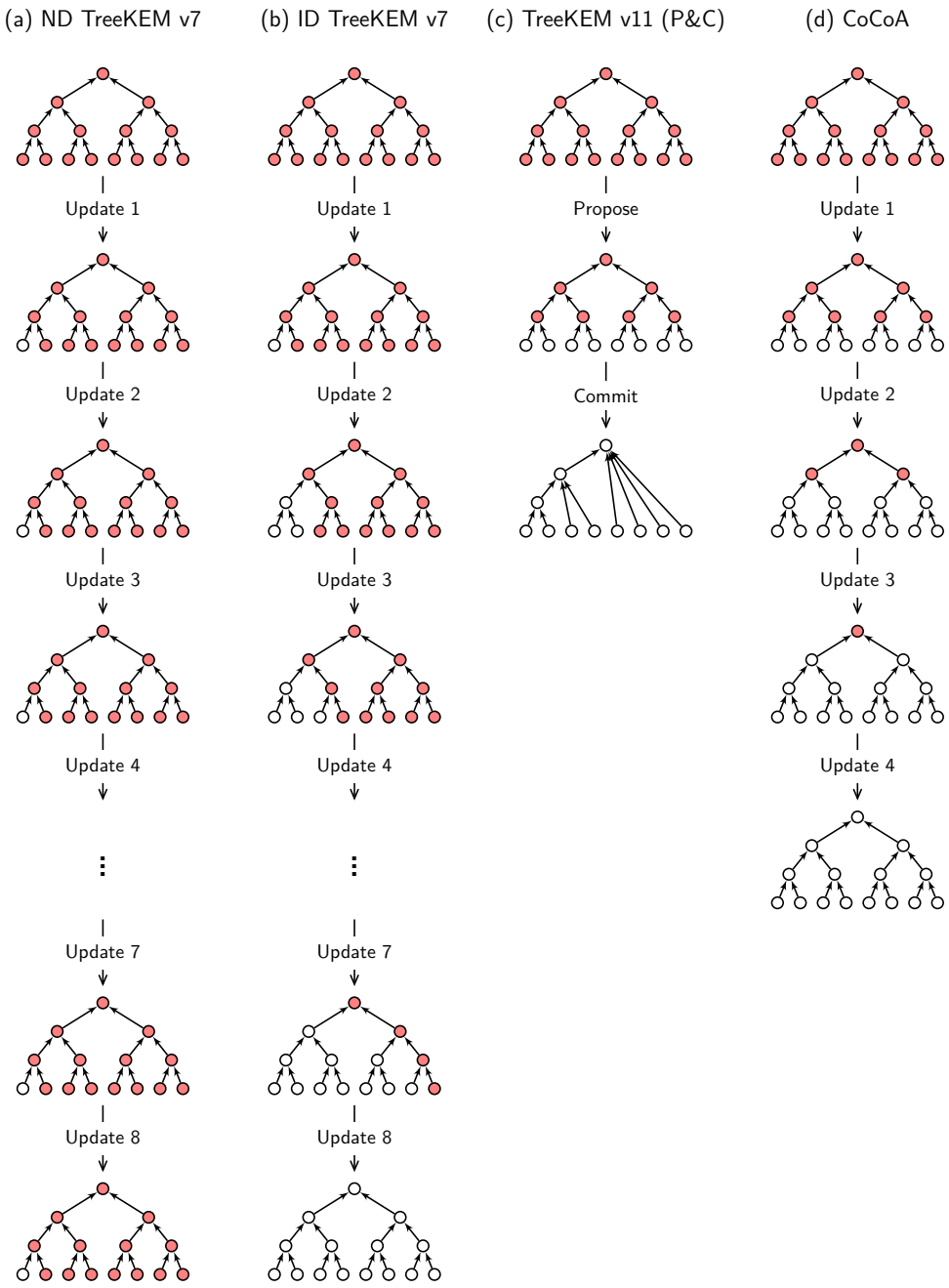

Figure 5.1: Comparison of number of rounds required to recover from corruption for different TreeKEM variants, ND stands for "Naïve Delivery", ID for "Ideal Delivery". Red nodes indicate key material known to the adversary. In each round all parties (try to) update. In columns (a) and (d) update requests are prioritized from left to right. In column (b) update requests are prioritized from left to right among all parties that did not update yet. In column (c) all parties propose an update, then the leftmost party commits.

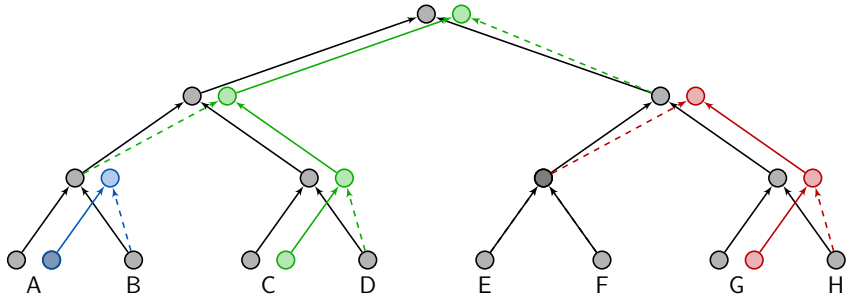

Figure 5.2: Example; concurrent updates in the CoCoA protocol. The former state of the ratchet tree (black) is changed by concurrent updates of A (blue), C (green), and G (red). The ordering is $U_C \prec U_A \prec U_G$. In the updates solid edges correspond to seeds obtained by hashing, dashed edges to encryptions.

As in MLS, it is assumed that parties have a public (list of) key(s) available to all other users, termed *init keys*, which can be used to add them to groups.

Removes can also be seen as a two-round process, as the removal of a party will cause v_{root} to get blanked, and an extra round will be needed to create a new group key - this is the same dynamic already existing in TreeKEM versions up to v7. Note that while this does not apply to later versions, these do need two rounds: one for the removal to be proposed, and another for it to be committed.

Note that concurrent operations could be conflicting: consider the case of two parties removing each other, a removed party adding a new one, or a removed party updating. Thus, special care needs to be taken in how to handle these conflicts. We refer the reader to Section 5.3.5 for more details.

Saving on communication complexity. A consequence of allowing concurrent operations is that arbitrarily many keys can change in a given round (even all the keys in the tree if all parties decide to update). In previous protocols, like MLS, every user stores a complete copy of the current ratchet tree. Sticking to this principle while allowing concurrent updates would imply that the communication cost of a round could be linear, as a party would have to download all new public keys in the tree. To avoid this, in the CoCoA protocol users only keep track of the state of nodes that are relevant to them when issuing an operation: those in their path and in the resolution of nodes in their co-path, since the latter will be the ones they need to encrypt to when sending an update. As an important consequence, the server no longer only acts as a relay server broadcasting the same message to all users. Instead, given a set of update, add, and remove operations, it prepares an individual packet for every user. We discuss the efficiency benefits of this approach in more detail in Section 5.4.[5]

[5]One could also imagine a variant of CoCoA that sticks closer to the principles of TreeKEM v11, by having the server forward complete update messages. While in this case t concurrently updating

Authenticity. Naturally, protocol messages have to be authenticated in any real-world deployment. In the MLS protocol, all users keep track of signature verification keys of the other users, and sign the update, add, and remove messages that they generate. As these messages are no longer simply forwarded by the server, in the CoCoA protocol users sign every component of the message separately. Removes and adds (and therefore initialization messages) consist of a single block of information that everyone needs to receive, so a single signature suffices. In case of an update, on the other hand, every ciphertext, and public key needs to be signed individually. However, this increase in computational cost and size of sender packets allows us to greatly decrease recipient packets in the way discussed above.

Robustness. An important property CGKA protocols aim for is robustness: ensuring that parties have consistent views of the tree. TreeKEM achieves what [AJM22] refers to as *weak robustness*: all honest parties accepting some message M (potentially generated adversarially) will transition to compatible states.[6]

To capture the CoCoA protocol, the definition of weak robustness needs to be slightly adapted: parties now receive different personalized packages as opposed to a unique one that gets broadcast to everyone, and do not have access to the complete ratchet tree. Accordingly, we require that if two parties receive and accept messages M_i and M_j satisfying a certain relation, they will transition into consistent states (where malformed messages not satisfying this relationship will immediately force users into inconsistent states). TreeKEM achieves weak robustness through a value called *confirmation tag* [AJM22]. This consists of a MAC of the entire CGKA transcript (encoded in a running hash, called the *transcript hash*) up to and *including* that epoch, which is sent together with every Commit message. The MAC key, a.k.a. the *confirmation key*, is derived from the new epoch key schedule, which ensures correct processing of the commit message and also that the sender had knowledge of the previous epoch's key schedule. To ensure consistency, users compute the transcript hash locally and verify the MAC. There are two issues when attempting to apply this to our scheme: 1) a user issuing an operation at a given round n will not have knowledge of the operations taking place concurrently, and thus will not be able to pre-compute the resulting transcript hash at the moment of crafting their message. And 2), since users only have a partial view of the ratchet tree, they are not able to compute the transcript hash. Note that users *need* to ensure they received consistent sets of operations, as e.g. in Figure 2, if

users would lead to a recipient complexity of $t \cdot \log(n)$, the protocol would still allow for concurrent updates that preserve the binary structure of the ratchet tree and thus outperform TreeKEM v11 for certain sequences of operations at the cost of slower healing.

[6] We mention in passing that [AJM22] also defines a notion of *strong robustness*, seemingly hard to achieve using practically efficient protocols, so we do not discuss it further.

C is not sent A's partial update, they will disagree on the key for node $\mathrm{Int}(A, B)$ after processing.

We solve 1) by effectively only authenticating the transcript up to the last round, i.e. not including the current operations. This ensures that if ID accepts a packet, it comes from a user whom they agreed with up until the beginning of that round. We solve 2) by what we call a *round hash*: a hash value computed over the public part of the new state of the ratchet tree (and any add and remove operations applied concurrently in that round). Clearly, none of the users can compute this hash value from their local state, so we shift this computation to the delivery server, who sends the round hash value to every party. However, the delivery server might act maliciously, and hence we need to ensure that the users can verify this computation. We do this by letting the round hash be a Merkle commitment to the current state. Users then expect the server to provide the openings of the commitment necessary to verify that it matches their partial view of the tree, which ensures consistency. See Section 5.3.3 for details.

Parent hash. TreeKEM aims for further security through two different mechanisms, mainly targeted at allowing new group members to verify the legitimacy of the received information. Informally, the first one, *parent hashing*, provides newly added users with the guarantee that the ratchet tree was well formed, i.e., that it resulted from a legitimate execution of the protocol. The second, *tree hashing*, introduced in TreeKEMv9 [BBR$^+$23], consists of a locally-computed hash commitment to the whole ratchet tree, which the new users can verify upon joining. Our round hash can be seen as subsuming it, just with the difference that it can no longer be computed locally.

We adapt the parent hash construction from TreeKEM into our protocol, with some modifications. While TreeKEM relies on parent hash to ensure new users can verify the tree received when joining a group, we also use it to verify any new keys sent by the server to current users. This is needed since, as a result of blanks, the set of nodes whose states users need to keep track of varies.

5.3.2 Users' states and the Key Schedule

Each user keeps track solely of the state of nodes on either their path or the resolution of their co-path; we define $\mathcal{P}(\mathrm{ID}) = \mathrm{path}(\mathrm{ID}) \cup \mathrm{Res}(\text{co-}\mathrm{path}(\mathrm{ID}))$ to be the set comprising exactly those nodes. More in detail, each user stores a local state γ, described in Table 5.2, which gets updated after every round message. We will write γ^n to refer to a state corresponding to round n.

$\gamma.\mathrm{ID}$	An identifier for the party.
$\gamma.G$	The set of current members of the group.
$\gamma.\mathrm{ssk}$	The party's signing key
$\gamma(v)$	Node state for every $v \in \mathcal{P}(\gamma.\mathrm{ID})$, only public part for $v \in \mathrm{Res}(\text{co-path}(\gamma.\mathrm{ID}))$.
$\gamma.\mathcal{H}_{trans}$	Current value of the transcript hash.
$\gamma.\mathrm{appSecret}$	Current round's application secret.
$\gamma.\mathrm{confKey}$	Current round's confirmation key.
$\gamma.\mathrm{initSec}$	Current round's initialization secret.
γ'	Pending state encoding operations not yet confirmed.

Table 5.2: User's local state γ.

Key schedule. CoCoA's *key schedule* for round n is defined via hash function H_5 as follows:

$$\begin{aligned}
\gamma.\mathrm{epochSecret}(n) &= \mathsf{H}_5(\gamma.\mathrm{initSec}(n-1) \,||\, \Delta_{root}(n) \,||\, \mathcal{H}_{trans}(n)) \\
\gamma.\mathrm{appSecret}(n) &= \mathsf{H}_5(\gamma.\mathrm{epochSecret}(n) \,||\, \texttt{'appsecret'}) \\
\gamma.\mathrm{confKey}(n) &= \mathsf{H}_5(\gamma.\mathrm{epochSecret}(n) \,||\, \texttt{'confirm'}) \\
\gamma.\mathrm{initSec}(n) &= \mathsf{H}_5(\gamma.\mathrm{epochSecret}(n) \,||\, \texttt{'init'})
\end{aligned}$$

The epoch secret $\gamma.\mathrm{epochSecret}(n)$ is used to derive all other keys from it; the application secret $\gamma.\mathrm{appSecret}(n)$ serves as the group key in epoch n and is to be used in higher level protocols, e.g. secure group messaging; the confirmation key $\gamma.\mathrm{confKey}(n)$ will be used to authenticate next epoch's protocol messages through a MAC termed the confirmation tag;[7] and the initialization secret $\gamma.\mathrm{initSec}(n)$ seeds next round's key schedule, tying it to the current one. Finally, the transcript hash $\mathcal{H}_{trans}(n)$ encodes the transcript of the execution up until round n - it is defined in the following section.

5.3.3 Robustness, Round Hash, and Transcript Hash

In this section we discuss CoCoA's robustness. We show that two parties accepting messages containing the same round hash value, will transition into consistent states. We start by defining the concept of a round hash and consistent states.

In the following we assume a fixed rule, that can be locally computed by the users on input a ratchet tree $\mathcal{T}$ and a set of operations that determines a total ordering of said

[7] This MAC, also present in TreeKEM, is there to mitigate active attacks. The latter are not reflected in our security model, but we chose to keep it, as it is the main security mechanism in response to a leaking of signature keys.

operations. This ordering ensures all users will compute the same round hash and also, when applied to adds A_i, determines the free leaf that the user added by A_i is assigned to.

Defintion 5.3.1. *Let* H_3 *be a hash function, and* n *a round with associated protocol messages* $T = (U, R, A) = ((U_1, \ldots, U_k), (R_1, \ldots, R_l), (A_1, \ldots, A_m))$*, where the* U_i *correspond to update messages; and the* R_i *and* A_i *correspond to the packets, as sent by their issuers, of any remove and add operation, respectively; and let each vector* U, R, A *be ordered with respect to the ordering* $\prec$*. Let* $\mathcal{T}^n$ *be the ratchet tree resulting from applying the operations in* T *with respect to* $\prec$ *to* $\mathcal{T}^{n-1}$*, and* ${}^p\gamma(v)$ *the public state of* v *in* $\mathcal{T}^n$ *(note that* ${}^p\gamma(v) = \texttt{blank}$ *if the node is to be blanked as a result of some removal in* R*). We define the map* ℓ *taking nodes in* $\mathcal{T}^n$ *to labels as follows:*

$$\ell(v) = \begin{cases} \mathsf{H}_3({}^p\gamma(v)), & \text{if } v \text{ is a leaf.} \\ \mathsf{H}_3(\ell(\mathrm{lparent}(v)), \ell(\mathrm{rparent}(v)), {}^p\gamma(v)), & \text{if } v \text{ is an internal node.} \end{cases}$$

The round hash $\mathcal{H}_{round}(n)$ *of* n *is defined to be*

$$\mathcal{H}_{round}(n) = \mathsf{H}_3\left(\ell(v_{root}), R, A\right) \quad .$$

In short, the round hash is essentially a Merkle commitment to the ratchet tree's public keys and the round's dynamic operations. The benefit of this approach is that every user can verify that the round hash sent by the server faithfully encodes the operations affecting their local state, by just receiving at most a logarithmic number of values irrespective of the number of updates (note that a user will necessarily need to hear about all dynamic operations). In particular, a user ID receiving the appropriate group operations should have access to the inputs corresponding to dynamic operations, and to the new keys of nodes in $\mathcal{P}(\mathrm{ID})$. The server does this by sending the user $\mathcal{H}_{round}(n)$, as well as the output of openRH (Figure 5.3), which, on input a user ID, returns a vector of hash values, corresponding to the labels of nodes not in $\mathcal{P}(\mathrm{ID})$, but that are parents of a node in $\mathcal{P}(\mathrm{ID})$. Given these values, the user is able to verify the received round message by running $\mathrm{verifyRH}$ (Figure 5.3), which recomputes $\mathcal{H}_{round}(n)$ with respect to their updated ratchet tree and compares it to the round hash provided by the server. In order to formally define the above algorithms, we make use of the following helper functions:

- $\mathrm{extract}$ on input a round message M and a state γ, outputs a list of updates $(U_1, \ldots, U_p)$, removes $(R_1, \ldots, R_q)$, and adds $(A_1, \ldots, A_s)$, a list of indices i corresponding to the leaves users added by the A_i are assigned to, and a round hash value h.

- update-info on input vectors U, R, A of update, remove and add operations, ordering $\prec$, a state γ corresponding to ID and a node $v \in \mathcal{P}(\mathrm{ID})$, outputs a node public state ${}^{p}\gamma_v$ corresponding to the public state of v resulting from applying the input operations with respect to $\prec$ and state γ (${}^{p}\gamma_v = \texttt{blank}$ if v is blanked as a result of some R_i); and, if v is a leaf node, additionally, outputs the identifier ID^* corresponding to v after applying the operations as above.

- retrieve-labels on input a local state γ, a node v and a vector O of tuples of the form (v_i, h, h'), outputs vector (v, h_l, h_r) if there is a unique vector in O with $v_i = v$, where $h_l = h$ and $h_r = h'$, and $\perp$ otherwise.

- tree on input a set of nodes V outputs the smallest subtree of $\mathcal{T}$ which contains V.

- depth on input a local state γ and a node v, outputs the depth of v, as defined by the length of the path from v to v_{root} in $\mathrm{tree}(\mathcal{P}(\gamma.\mathrm{ID}))$.

Correctness of algorithm $\mathrm{verifyRH}$ follows by inspection. The following lemma shows that we get the desired robustness. Note that if two users process a round message containing different round hash values, they will immediately be forced into inconsistent states, so we can just concern ourselves with the case where the round hash values are the same.

The transcript hash is defined as $\mathcal{H}_{trans}(0) = 0$, and, for subsequent rounds, given a verified round hash:

$$\mathcal{H}_{trans}(n) = \mathsf{H}_3(\mathcal{H}_{trans}(n-1)||\mathcal{H}_{round}(n)) \ .$$

With this we can define what it means for parties to have consistent states, which informally requires them to have consistent views of the tree (i.e. agree on the states of nodes on the intersection of their states), and agree on the group key, group members, and group history, i.e. on the transcript hash.

Defintion 5.3.2. *Let* ID *and* ID^* *be two group members with states* γ *and* γ^**. They have consistent states if* ${}^{p}\gamma(v) = {}^{p}\gamma^*(v)$ *for all* $v \in \mathcal{P}(\mathrm{ID}) \cap \mathcal{P}(\mathrm{ID}^*)$*,* $\gamma.\mathrm{appSecret} = \gamma^*.\gamma.\mathrm{appSecret}$*, and* $(\gamma.G, \gamma.\mathcal{H}_{trans}) = (\gamma^*.G, \gamma^*.\mathcal{H}_{trans})$.

Note that we only define consistency of states for users who have joined the group. More in detail, we say that a user ID has (in their view) joined the group if there exists a query $\mathrm{CGKA.Proc}(\mathrm{ID}, \cdot)$ in the execution, where ID accepts the corresponding round message, i.e. where the state for ID changes (is initialized) as a result of said query.

Algorithm $\mathrm{openRH}(\mathrm{ID}, \mathcal{T}_\ell)$

$O := ()$
For $v \in \mathrm{Res}(\text{co-path}(\mathrm{ID}))$**:**
  If $v.\mathrm{isLeaf} =$ **false:**
    $O \leftarrow O \cup (v, \ell(\mathrm{lparent}(v)), \ell(\mathrm{rparent}(v)))$
Return O

Algorithm $\mathrm{verifyRH}(\gamma, M, O)$

$(U, R, A, \mathsf{i}, h) \leftarrow \mathrm{extract}(M, \gamma)$
For $d \in \{\mathrm{depth}(\mathrm{leaf}(\gamma.\mathrm{ID})), \ldots, 0\}$**:**
  For $v \in \mathrm{tree}(\mathcal{P}(\gamma.\mathrm{ID}))$ **s.t.** $\mathrm{depth}(v) = d$**:**
    If $v.\mathrm{isLeaf} =$ **true:**
      ${}^p\gamma_v \leftarrow \text{update-info}(U, R, A, \prec, \gamma, v)$
      $\ell(v) = \mathsf{H}_3({}^p\gamma_v)$
    Else If $v \in \mathrm{Res}(\text{co-path}(\mathrm{ID}))$**:**
      ${}^p\gamma_v \leftarrow \text{update-info}(U, R, A, \prec, \gamma, v)$
      $(v, h_l, h_r) \leftarrow \text{retrieve-labels}(O, \gamma, v)$
      $\ell(v) = \mathsf{H}_3(h_l, h_r, {}^p\gamma_v)$
    Else
      ${}^p\gamma_v \leftarrow \text{update-info}(U, R, A, \prec, \gamma, v)$
      $\ell(v) = \mathsf{H}_3(\ell(\mathrm{lparent}(v)), \ell(\mathrm{rparent}(v)), {}^p\gamma_v)$
If $h = H(\ell(v_{root}), R, A, \lambda)$**:**
  $b \leftarrow 1$
Else $b \leftarrow 0$
Return b

Figure 5.3: Round Hash algorithms

The following proposition shows that we get the desired robustness. Note that if two users process a round message containing different round hash values, they will immediately be forced into inconsistent states, so we can just concern ourselves with the case where the round hash values are the same.

Proposition 1. *Let* ID_i *and* ID_j *be two group members with consistent local state after round* n *and let them receive round messages* M_i *and* M_j *respectively, both containing the same round hash value* h*, and opening vectors* O_i *and* O_j*. If* $\mathrm{verifyRH}(\gamma_i^n, M_i, O_i) = \mathrm{verifyRH}(\gamma_j^n, M_j, O_j) = 1$*, then they will have consistent states after processing their respective round messages.*

Proof. Assume for contradiction that the parties arrive to inconsistent states after

processing their respective messages, we will show that this implies a collision for H. Since we assumed them to start the round in a consistent state, and the received round hash is the same, they will agree on the transcript hash. Similarly, agreement on the group ID and round follows trivially from the agreement in the previous round. Thus, disagreement over the group state must come from either membership or key material in the tree. First, let $(U_k, R_k, A_k, \prec_k, \lambda_k, h) \leftarrow \mathrm{extract}(M_k, \gamma_k)$ for $k \in \{i, j\}$ and denote by ℓ_k the map from nodes to labels as derived by party ID_k when running algorithm $\mathrm{verifyRH}$. Define $\alpha_k = (\ell_k(v_{root}), R_k, A_k, \lambda_k)$. We know from $\mathrm{verifyRH}(\gamma_i^n, M_i, O_i) = \mathrm{verifyRH}(\gamma_j^n, M_j, O_j) = 1$ that $H(\alpha_i) = H(\alpha_j)$. Now, if $\gamma_i^{n+1}.G \neq \gamma_j^{n+1}.G$, it follows that $(R_i, A_i) \neq (R_j, A_j)$, since $\gamma_i^n.G = \gamma_j^n.G$. In particular, this implies that $\alpha_i \neq \alpha_j$, i.e. a collision for H. Assume, therefore, that $\gamma_i^{n+1}.G = \gamma_j^{n+1}.G$ and that there is $v \in \mathcal{P}(\mathrm{ID}_i) \cap \mathcal{P}(\mathrm{ID}_j)$ such that ${}^p\gamma_i^{n+1}(v) \neq {}^p\gamma_j^{n+1}(v)$. If $h_i \neq h_j$, a collision for H follows as before, so assume that $h_i = h_j$. Let u, u' be the left and right parents, respectively, of v_{root}. By definition, we have that $h_k = H(\ell_k(u), \ell_k(u'), {}^p\gamma_k^{n+1}(v_{root}))$. If any of $\ell_i(u) \neq \ell_j(u)$, $\ell_i(u') \neq \ell_j(u')$ and ${}^p\gamma_i^{n+1}(v_{root}) \neq {}^p\gamma_j^{n+1}(v_{root})$ hold, a collision for H follows. Assume thus equality of the labels and public states, and assume w.l.o.g that v is on the left subtree, i.e. is an ancestor of u. We know that $\ell_i(u) = \ell_j(u)$, and that $\ell_k(u) = H(\ell_k(w), \ell_k(w'), {}^p\gamma_k^{n+1}(u))$, by definition, where w_k, w_k' are the left and right parents of u. Thus, as before, either we have some inequality between the corresponding values for i and j, from which a collision can be extracted, or the labels for the parents are the same for both parties. We can repeat this process until arriving to v, for which we know ${}^p\gamma_i^{n+1}(v) \neq {}^p\gamma_j^{n+1}(v)$. We will therefore eventually find a collision for H.

Therefore, an adversarial delivery server making ID_i and ID_j accept messages with a consistent round hash value, which prompt them into inconsistent states would equivalently be able to derive a collision for H. Since we assume H to be collision resistant, it follows that no such adversary exists. □

The next proposition shows that, in fact, it is to consider the group keys or transcript hash values in users states to determine if these are consistent.

Proposition 2. *If* H_3 *and* H_5 *are modeled as random oracles then, with all-but-negligible probability, the states* γ *and* γ^* *of users* ID *and* ID^* *are consistent if and only if* $\mathrm{CGKA.Key}(\gamma) = \mathrm{CGKA.Key}(\gamma^*)$. *Equivalently, if they have the same transcript hash value* $\gamma.\mathcal{H}_{trans} = \gamma^*.\mathcal{H}_{trans}$.

Proof. The only if implication is clear by definition. Assume thus that $\mathrm{CGKA.Key}(\gamma) = \mathrm{CGKA.Key}(\gamma^*)$. Since the application secret in their states is the same, so must be $\mathcal{H}_{trans}$, by collision resistance of H_5. If the states of the two parties differ in the group membership, then one of them must have processed some add or remove operation the

other has not. These operations are directly hashed into the appropriate round hash, which itself gets hashed into $\mathcal{H}_{trans}$. Thus, there must have been a collision in either H_3 or H_5. Finally, the public states for all nodes in a user's state get hashed by the user in order to compute the round hash. As before, if any of the values in the states of nodes in the intersection of user's states $\mathcal{P}(\mathrm{ID}) \cap \mathcal{P}(\mathrm{ID}^*)$ differs, we can extract a collision for either H_3 or H_5. Hence, the states of the two users must be consistent.

Finally, to see that two users having the same transcript hash value implies them having the same epoch secret $\mathrm{CGKA.Key}(\gamma)$, recall that $\gamma.\mathrm{epochSecret}(n) = \mathsf{H}_5(\gamma.\mathrm{initSec}(n-1) \,||\, \Delta_{root}(n) \,||\, \mathcal{H}_{trans}(n))$, where $\gamma.\mathrm{initSec}(n) = \mathsf{H}_5(\gamma.\mathrm{epochSecret}(n) \,||$ 'init'). Moreover, a user always checks that the secret key at any node v, and in particular at v_{root}, is consistent with the public key set at that node. The former is derived from Δ_v, and the latter is input into the computation of the round hash, and therefore of $\mathcal{H}_{trans}$. Thus, were the seeds $\Delta_{root}(n)$ used by ID and ID^* to compute their respective epochs secrets different, so would be their transcript hash values, up to the negligible probability of collisions of $\mathrm{PKE.Gen}$ and the random oracle H_2 (since, recall, $(\mathrm{sk}_i, \mathrm{pk}_i \leftarrow \mathrm{PKE.Gen}(\mathsf{H}_2(\Delta_i)))$. Last, suppose for contradiction that the initialization secrets are different while their transcript hash values match. Since the $\gamma.\mathrm{initSec}$ are derived deterministically from $\gamma.\mathrm{epochSecret}$, if $\gamma.\mathrm{initSec}(n-1) \neq \gamma.\mathrm{initSec}^*(n-1)$, then we would have $\gamma.\mathrm{epochSecret}(n-2) \neq \gamma.\mathrm{epochSecret}^*(n-2)$. By the collision resistance of the random oracle H_3, $\mathcal{H}_{trans}(n) = \mathcal{H}_{trans}{}^*(n)$ implies $\mathcal{H}_{trans}(n-1) = \mathcal{H}_{trans}{}^*(n-1)$, so we could apply the same argument, eventually reaching the first epoch where the last one of them joined, i.e. to the first point in the execution where $\mathcal{H}_{trans} = \mathcal{H}_{trans}{}^*$. If at this point one of the users, say w.l.o.g. ID, was already a part of the group (i.e. this was not the value with which one of them initialized their state), then ID^* got both their $\gamma.\mathrm{initSec}^*$ and $\mathcal{H}_{trans}{}^*$ from another party ID', so we can repeat the argument with respect to ID and ID'. If, instead, these values corresponded to those with which both of them initialized their state, then by collision resistance, they must have been sent by different parties ID_1 and ID_2, for which the same must be true: their transcript hash values match, but not their initialization secrets. Eventually, since the number of users in the group is finite and there is a single initial user, the group creator, it must be that these two users are the same or were added by the same party with respect to the same state (by collision resistance of the random oracle H_3), which is a contradiction.

□

5.3.4 Parent Hash

Ratchet trees in TreeKEM contain so-called *parent hashes*, which were introduced to the standard in TreeKEM v9, and analyzed and improved by Alwen *et al.* [AJM22]. These ensure, on the one hand, that for every node $v \in \mathcal{T}$, whoever sampled sk_v had

knowledge of the secret signing key for some leaf l of the subtree rooted at v; and on the other, that at the moment this secret was generated it was not communicated to any user whose leaf is not in this subtree. This protects against active attacks where a user is added to a malformed group where the tree invariant is violated, potentially causing him to communicate to a set of users different to the one he believes to be communicating to.

To adapt parent hash to CoCoA we have to overcome the two issues that (a), since parties update concurrently, parent hash values can be defined with respect to keys on the copath that were overwritten by a concurrent update, and (b), since the resolution of a user's copath and in turn the corresponding public keys that are known to the user may change from round to round, the user needs to be able to verify the authenticity of such keys without having access to the state of leaves below it. We address the first issue by having users store the public keys of one previous round: each node state $\gamma(v)$ now contains an associated list of predecessor keys, $\mathrm{PK}^{\mathrm{pr}}$, containing the public keys corresponding to nodes in the resolution of v in the epoch when the current key was sampled, and excluding those that where unmerged at $\mathrm{child}(v)$;[8] i.e. if the update sampling pk_v unblanked v, the predecessor keys will be a list, else it will just contain the previous public key. The second issue we solve by not only signing the parent hash value of users' leaves but by introducing a signature at every node in their update path (that which is sent with the packet containing the new public key when it is first announced). Last, to ensure consistency between users' views, we add two further values to the parent hash and node state: a commitment to the subtree under the node's sibling and a commitment to the whole ratchet tree. We now define more formally the slightly modified parent-hash algorithm, compatible with our construction, with respect to signature scheme SIG.

As in TreeKEM, parent hash values of a node are updated whenever the key corresponding to the node is updated. More in detail, let ID compute an update U containing new keys for nodes along their path (see full definition in Section 5.3.5), which get stored in pending state γ'. Parent hashing algorithm $\mathrm{PHash.Sig}$ on input (ID, γ') first fetches ID's update path $\mathrm{path}(v_{\mathrm{ID}}) = (v_0 = v_{\mathrm{ID}}, v_1, \ldots, v_k = v_{root})$. For $i \in \{0, \ldots, k-1\}$ let v'_i denote the parent of v_{i+1} that is not part of $\mathrm{path}(v_{\mathrm{ID}})$, and let $\mathcal{R} = \mathrm{Res}(v'_i) \setminus \mathrm{Unmerged}(v_{i+1})$. Then, we define $h_{1,k} = h_{2,k} = 0$, and using hash function H_4, compute:

[8] The exclusion of these unmerged leaves responds to the fact that these could correspond to parties added *after* the state for $\mathrm{child}(v)$ was last updated.

$$h_{1,i} \leftarrow \ell(v'_i) \qquad \text{for } i \in (k-1,\ldots,0)$$
$$h_{2,i} \leftarrow \mathsf{H}_4(\mathrm{pk}_{v_{i+1}}, \mathrm{PK}^{\mathrm{pr}}_{v_{i+1}}, h_{2,i+1}, \{\mathrm{pk}_v\}_{v\in\mathcal{R}}) \qquad \text{for } i \in (k-1,\ldots,0)$$
$$\sigma_i \leftarrow \mathrm{SIG.Sig}\left(\gamma(\mathrm{ID}).\mathrm{ssk}, (\mathrm{pk}_{v_i}, \mathrm{PK}^{\mathrm{pr}}_{v_i}, (h_{1,i}, h_{2,i}), \mathcal{H}_{trans}, \mathrm{confTag})\right) \qquad \text{for } i \in (0,\ldots,k)$$

where $\ell(v)$ is the label of v as in Def. 5.3.1 above, $\mathrm{PK}^{\mathrm{pr}}_v \leftarrow 0$ if v did not have a key before U, $h_i = (h_{1,i}, h_{2,i})$.

Algorithm $\mathrm{PHash.Sig}$ then adds the values $(H, \Sigma) = (h_0, \ldots, h_k, \sigma_0, \ldots, \sigma_k)$ to U, substitutes the parent hash values h_i and signatures σ_i in γ' by the newly computed ones, and returns U.

Verification. A user receiving a tree T from the server can verify its authenticity by running the algorithm $\mathrm{PHash.Ver}(T)$. This will be run by users in two different scenarios: on the one hand, when joining the group, they will verify the whole ratchet tree (in this case $T = \mathcal{T}$); on the other, when processing a round message containing one or more removes, they will verify the received keys for nodes in the new resolution of their co-path (in this case T is the union of $\mathcal{P}(\mathrm{ID}_i)$ for all removed ID_i). The algorithm runs as follows:

The algorithm first checks that all non-blank nodes in the tree have a complete public state, and that for any internal node v, the associated identifier ID_v is associated to one of the leaves of the sub-tree rooted at v.[9] If any of these checks does not pass, the algorithm aborts. Next, it checks that $h_{2,v_{root}}$, and then, verifies the following conditions hold:

1. For any non-blank non-leaf node v in T the following equalities hold with either p and p' being the left and right parents of v or, if not, with p' being the left parent of v and p the right parent, setting $p \leftarrow \mathrm{lparent}(p)$ if p is blank, until p is either non-blank or an empty leaf, in which case $0 \leftarrow \mathrm{PHash.Ver}(T)$.[10]

(a) $h_{2,p} = \mathsf{H}_4(\mathrm{pk}_v, \mathrm{PK}^{\mathrm{pr}}_v, h_{2,v}, \{\mathrm{pk}_w\}_{w\in R})$ and $h_{1,p} = \ell(p')$ or

(b) $h_{2,p} = \mathsf{H}_4(\mathrm{pk}_v, \mathrm{PK}^{\mathrm{pr}}_v, h_{2,v}, \mathrm{PK}^{\mathrm{pr}}_{p'})$ and $\mathcal{H}_{trans,p} = \mathcal{H}_{trans,p'}$.

where $R = \mathrm{Res}(p') \setminus \mathrm{Unmerged}(v)$.

[9] A user who is already part of the group will have knowledge of the leaf index of each group member, and can check this without necessarily having a full view of the tree.

[10] The recursion in the second case is needed to account for the possible blank nodes introduced between p and v as a result of adding to new leaves to accomodate new parties, so that p and p' correspond to the parents of v at the time the state of v was created.

2. $\mathrm{SIG.Ver}_{\mathrm{svk}_w}((\mathrm{pk}_w, \mathrm{PK}_w^{\mathrm{pr}}, h_w, \mathcal{H}_{trans,w}, \mathrm{confTag}_w), \sigma_w) = 1$ for all $w \in T$.

We say that p and p' as defined above are the *effective parents* of v. If p is the effective parent satisfying condition 1, we say that v *verifies through* p, and note that this corresponds to the situation where p's key was sampled in the same update as v's. Observe that if we have two concurrent updates, the parent hash value of the node at the intersection cannot possibly have been computed with respect to the new key in its co-path, as these were generated at the same time, by different parties. Thus, the parent hash value will be computed w.r.t. the predecessor key - condition (b) corresponds to this case. Moreover, we check that in this case the users computing the concurrent updates where actually in consistent states, by checking that transcript hash values they included in their update packets are the same. In case v is not the intersection between two concurrent updates, the public state of the nodes in the resolution of the co-parent will not have changed, so we only need to check with respect to the newest key in the resolution nodes. However, we also check that the user who sampled v indeed had in their view the same (commitment to) subtree under the parent of v not in their path, by checking that the parent hash value h_1 is consistent with the label of p'. Finally, unmerged leaves are excluded from R since these might differ between the time the parent hashes were computed and the new user joins the group.

5.3.5 The Protocol: CoCoA and Partial Updates

In the description below, we use γ for the state of the party issuing the appropriate operation. The ordering used to resolve conflicts caused by concurrent updates is denoted by $\prec$. An overview of the content of user-generated messages can be found in Table 5.3, and one for the contents of round messages in Table 5.4.

Initialization. To initialize a group with parties $G = \{\mathrm{ID}_1, \ldots, \mathrm{ID}_n\}$, ID_1 creates a ratchet tree as follows. First, ID_1 retrieves the public initialization keys $(\bar{pk}, \bar{svk}) = (\{\mathrm{pk}_{\mathrm{ID}_1}, \ldots, \mathrm{pk}_{\mathrm{ID}_n}\}, \{\mathrm{svk}_{\mathrm{ID}_1}, \ldots, \mathrm{svk}_{\mathrm{ID}_n}\})$ of all group members (including themselves), redefines $G \leftarrow (G, \mathrm{pk}, \mathrm{svk})$ to include these, and initializes a left-balanced binary tree with n leaves, assigning each pair of keys in $(\bar{pk}, \bar{svk})$ to a leaf. Let v be ID_1's leaf. They then sample new secrets for v's path $(\Delta, K) \leftarrow \text{re-key}(v)$, store the new keypairs $(\mathrm{sk}_j, \mathrm{pk}_j)$ in the corresponding nodes on the created tree and compute and store in γ' the parent hashes and signatures for the nodes in $\mathrm{path}(v)$: $(H, \Sigma) \leftarrow \mathrm{PHash.Sig}(\mathrm{ID}_1, \gamma')$, where recall that each $\sigma_j \in \Sigma$ is a signature of $(\mathrm{pk}_j, 0, h_j, \mathcal{H}_{trans}, 0)$ for some $v_j \in \mathrm{path}(v)$ with $h_j \in H$ its corresponding new parent hash pair (here $\mathrm{PK}_v^{\mathrm{pr}}$ and $\mathrm{confTag}$ are set to 0 initially). For every $v_j \in \mathrm{path}(v) \setminus v$, let w_j be the parent of v_j not in $\mathrm{path}(v)$. Then, for each $y_{j,l} \in \mathrm{Res}(w_j)$, ID_1 computes $e_{j,l} = \mathrm{PKE.Enc}(\mathrm{pk}_{y_{j,l}}, \Delta_j)$, together with the signature $\sigma^s_{j,l} = \mathrm{SIG.Sig}_{\mathrm{ssk}_i}(e_{j,l})$. Next,

Init	
ID	The identifier of the group creator.
G	List of group members.
$P = (p_j = (\mathrm{pk}_j, h_j, \mathcal{H}_{trans}, \sigma_j) \ : \ j \in [\lvert\mathrm{path}(\mathrm{ID})\rvert])$	Public states for every $v_j \in \mathrm{path}(\mathrm{ID})$.
$S = (s_{j,l} = (e_{j,l}, \sigma^s_{j,l}) \ : \ j \in [\lvert\mathrm{path}(\mathrm{ID})\rvert],\ l \in L_j)$	Encryptions of the seeds along ID's path.
Update	
ID	The identifier of the updating user.
c_i	Update counter
$P = (p_j = (\mathrm{pk}_j, h_j, \mathcal{H}_{trans}, \mathrm{confTag}, \sigma_j) \ : \ j \in [\lvert\mathrm{path}(\mathrm{ID})\rvert])$	New public states for every $v_j \in \mathrm{path}(\mathrm{ID})$.
$S = (s_{j,l} = (e_{j,l}, \mathrm{confTag}, \sigma^s_{j,l}) \ : \ j \in [\lvert\mathrm{path}(\mathrm{ID})\rvert],\ l \in L_j)$	Encryptions of the new seeds.
Remove/Add	
ID	Identifier of removed/added party.
$\mathrm{confTag}$	Confirmation tag.
σ	Signature of message contents under sender's signing key.
$\mathrm{pk}_{\mathrm{ID}}$	Public key of added party (Only adds).
$\mathcal{W} = (\mathcal{H}_{round}, \gamma.\mathcal{H}_{trans}, \gamma.G, \gamma.\mathrm{confKey}, \gamma.\mathrm{initSec})$	Welcome message (Sent next round, signed, only adds).

Table 5.3: Contents of user generated messages.

they send out the initialization message $I = (\mathit{'init'}, \mathrm{ID}_1, G, \bar{pk}, P, S)$; where P is the vector with entries $p_j = (\mathrm{pk}_j, h_j, \mathcal{H}_{trans}, \sigma_j)$, one per node in $\mathrm{path}(\mathrm{ID}_1)$; and S is the vector with entries $s_{j,l} = (e_{j,l}, \sigma^s_{j,l})$, containing all the necessary encryptions and values to be authenticated, for each $v_j \in \mathrm{path}(v)$. Finally, ID_1 erases the seeds Δ_i, and sets all internal nodes outside their path in their local tree copy to be blank.

Update. To issue an update, user ID_i with state γ and at leaf v, first computes new secrets along their path $(\Delta, K) \leftarrow \text{re-key}(v)$, stores the new keys in γ' and computes and stores the parent hashes and signatures for the nodes in $\mathrm{path}(v)$: $(H, \Sigma) \leftarrow \mathrm{PHash.Sig}(\mathrm{ID}_i, \gamma')$. Second, they set $\mathrm{confTag} = \mathrm{MAC.Tag}(\gamma.\mathrm{confKey}, \gamma.\mathcal{H}_{trans})$. For every $v_j \in \mathrm{path}(v) \setminus v$, let w_j be the parent of v_j not in $\mathrm{path}(v)$ and let $L_j = \mathrm{Res}(w_j) \cup \mathrm{Unmerged}(\mathrm{Res}(w_j))$ be the set of nodes that are either in the resolution of w_j or are leaves that are unmerged at some node in said resolution. Then, for each $y_{j,l} \in L_j$, ID_i computes $e_{j,l} = \mathrm{PKE.Enc}(\mathrm{pk}_{y_{j,l}}, \Delta_j)$, together with the signature $\sigma^s_{j,l} = \mathrm{SIG.Sig}_{\mathrm{ssk}_i}(e_{j,l}, \mathrm{confTag})$. Next, they send out the update message $U = (\mathrm{ID}_i, P, S, c_i)$; where P is a vector of entries $p_j = (\mathrm{pk}_j, h_j, \mathcal{H}_{trans}, \mathrm{confTag}, \sigma_j,)$ containing the new public states and necessary authentication values for each $v_j \in \mathrm{path}(v)$;[11] S is the vector with entries $s_{j,l} = (e_{j,l}, \mathrm{confTag}, \sigma^s_{j,l})$, containing all the necessary encryptions and values to be authenticated, for each $v_j \in \mathrm{path}(v)$; and a counter c_i, the number of updates (including this one) sent by ID_i since they last processed a round message. Last, they erase the seeds Δ.

[11] note that, as in an initialization message, the signature included in each of the p_j does not exactly cover the rest of the elements of p_j, but also includes the predecesor key $\mathrm{PK}^{\mathrm{pr}}$ at that node. This is not a problem for verification, as this is set to 0 for new groups, and in any other cases, parties will have access to the key at that node before they processed said update.

Remove. To remove party ID_j, ID_i sends out a $\texttt{remove}(\mathrm{ID}_j)$ plaintext request together with $\mathrm{confTag} = \mathrm{MAC.Tag}(\gamma.\mathrm{confKey}, \gamma.\mathcal{H}_{trans})$, and a signature σ under their signing key of the remove message and the confirmation tag. This will have the effect of blanking the nodes in ID_j's path. Following a removal, an update operation must be issued immediately so that a new group key is created.

Add. Additions of parties work in two rounds. To add party ID_j, ID_i first sends a plaintext add request $\texttt{add}(\mathrm{ID}_j, \mathrm{pk}, \mathrm{svk})$ containing ID_j's public init key pair $(\mathrm{pk}, \mathrm{svk})$, $\mathrm{confTag} = \mathrm{MAC.Tag}(\gamma.\mathrm{confKey}, \gamma.\mathcal{H}_{trans})$ and a signature under ID_i's signing key of the add request and the confirmation tag. This will allow all group members to learn the identity of the new party and therefore to encrypt future protocol messages to them. In the following round, ID_i[12] must send ID_j a signed welcome message $\mathcal{W} = (\mathcal{H}_{round}, \gamma.\mathcal{H}_{trans}, \gamma.G, \gamma.\mathrm{confKey}, \gamma.\mathrm{initSec})$, encrypted under pk, containing the necessary information to initialize their local state and seed that round's key schedule, as well as check the received tree from the server is correct. Moreover, some user must send an update during that round, ensuring that way that a new application secret will be created.

Collect and Deliver. Whenever the server receives an initialization message I, it just forwards it to all the new group members, initializing its local state γ_{ser} with the members of the new group and the public information of the ratchet tree included in I. For all other messages, it does as follows: given concurrent group messages $T = (U, R, A, W) = (U_a, R_b, A_c, W_d : a \in [p], b \in [q], c \in [r], d \in [s])$ sent during a round, corresponding to updates, removes, adds, and welcome messages, respectively, the delivery server will first check if any two or more updates come from the same user, deleting all of them except for the one received last. The server first updates its local copy of the public state of $\mathcal{T}$, stored in γ_{ser}, by updating the public keys of nodes refreshed by any U_a, blanking any nodes affected by any R_b, and adding a public key and identifier to any leaf newly populated as a result of an A_c; here if two or more operations affect a given node, the operation that is minimal with respect to $\prec$ will be the one determining the state of the node. A few considerations must be observed here, which we discuss further below: first, all removes must precede any updates, so that a node is blanked whenever a leaf under it is removed, irrespective of which updates take place; second, conflicting removes take effect simultaneously, blanking nodes in both paths; third, new users are added on the left-most free leaves in the tree according to some fixed rule that the receiving parties can reproduce locally. Once the server's view $\mathcal{T}$ is updated, it computes the labels for it, defining $\mathcal{T}_\ell$, and the round hash $\mathcal{H}_{round}$, as prescribed in Definition 5.3.1; and computes opening vectors $O_i \leftarrow \mathrm{openRH}(\mathrm{ID}_i, \mathcal{T}_\ell)$ for all group members $\mathrm{ID}_i \in G$ (note that these will be computed with respect to the

[12] an alternative specification could allow any group member online to do this instead

R	Vector of remove operations
A	Vector of add operations
c_i	Update counter
$\mathcal{H}_{round}$	Round hash value
O_i	Openings to verify $\mathcal{H}_{round}$
$\gamma(v)$ for $v \in N_i$	public states of nodes needed to update $\mathcal{P}(\mathrm{ID}_i)$
$u_v = (\mathrm{ID}, p, s)$	public state and encryption (if appropriate) of each relevant updated v
$\mathcal{W}$	Welcome message (only for new joiners)
$\mathcal{T}$	Entire ratchet tree (only for new joiners)

Table 5.4: Contents of Round message M_i to party ID_i.

set $\mathcal{P}(\mathrm{ID}_i)$ resulting from (un)blanking nodes as implied by T). Then, it crafts round messages M_i for each user, containing the following information: first, the vectors R and A or removes and adds; second the vector O_i and the round hash $\mathcal{H}_{round}$; third, the public states $\gamma(v) = (\mathrm{pk}_v, \mathrm{PK}_v^{\mathrm{pr}}, h_v, \mathrm{ID}_v, \sigma_v, \mathcal{H}_{trans,v}, \mathrm{confTag}_v, o_v, \mathrm{Unmerged}(v))$ at the beginning of the round of the nodes $v \in \mathcal{N}_i = (\cup_{j \in R_{id}} \mathcal{P}(\mathrm{ID}_j)) \setminus \mathcal{P}(\mathrm{ID}_i)$ where R_{id} is the set of indices of parties removed by R, i.e., the new nodes on the resolution of ID_i's and the extra states needed to verify the validity of the received keys[13]; and fourth, for each node $v \in \mathcal{P}(\mathrm{ID}_i)$ (after the (un)blanking implied by T) whose keys get rotated as a result of some (winning w.r.t. $\prec$) update $U_a = (\mathrm{ID}, P, S)$, the server adds $u_v = (\mathrm{ID}, p_j)$ to M_i, where $p_j \in P$ is the public state of corresponding to v; if, besides, $v \in \mathrm{path}(\mathrm{ID}_i)$ and is the lowest node in $\mathrm{path}(\mathrm{ID}_i)$ updated by U_a, the server also includes the tuple $s_{j,l} \in S$ into u_v, corresponding to the encryption of v's seed to the node in $\mathrm{path}(\mathrm{ID}_i)$ which is in the resolution of the co-path of U_a's author. Last, the server also includes a counter c_i, equal to that of ID_i's update included in M_i if there is one, and 0 otherwise. Finally, for each newly-added ID_i, the round message M_i additionally contains the corresponding $\mathcal{W}$, as well as a copy of the public state of $\mathcal{T}$.

Process. Upon receipt of a round message M containing associated updates $U = (U_1, \ldots, U_p)$, removes $R = (R_1, \ldots, R_q)$, adds $A = (A_1, \ldots, A_r)$, openings vector O, public states for nodes in $\mathcal{N}$, round hash $\mathcal{H}_{round}$, and counter c, user ID processes it as follows. First, if $c \neq 0$, they check if, from the time they last processed a round message, they issued an update with counter c, aborting if not. Next, for every update, remove and add, they check that $\mathrm{MAC.Ver}(\gamma.\mathrm{confKey}, \mathrm{confTag}) = 1$; that for the all update packets U_a the transcript hash value included with the new public values for a node is the same as $\gamma.\mathcal{H}_{trans}$; and that the associated signature verifies under the public key of

[13] note that the leaves of the sub-tree of $\mathcal{T}$ with vertex set $\mathcal{N}_i$ correspond to the new nodes in the resolution of ID that were not part of their state

the sender (using the current node key in place of $\mathrm{PK}^{\mathrm{pr}}$ to verify signatures of updates); and similarly abort if any of these verifications does not pass. [14] If these checks pass, they copy their local state γ corresponding to the current round to γ', incorporating into it any node states previously stored there as part of the generation of said update with counter c (this update is empty if $c = 0$). Then, they update the public state of nodes needed to verify the round hash, as prescribed by the received operations: first, for every $v \in \mathcal{P}(\mathrm{ID})$, they blank v if it is in the path affected by some R_i and update $\mathcal{P}(\mathrm{ID})$ to include its new resolution as follows: they check that the set of nodes $\mathcal{N}$ consists of the nodes outside $\mathcal{P}(\mathrm{ID})$ that are in the paths and resolutions of co-paths of removed users. If more than one user is removed, it could be that $\mathcal{N}$ consists of several disconnected subtrees of $\mathcal{T}$. For each such subtree T, ID checks that its leaves are all non-blank; that all the leaves (w.r.t to $\mathcal{T}$) of removed parties as described in R are included in it; and, finally, that $\mathrm{PHash.Ver}(\gamma, T) = 1$. Moreover, for each blanked node w (as a result of M), they will use the received openings for the leaves of T, together with the received states, to reconstruct the Merkle hash openings o_v associated to v and check that the stored values match these. If all the checks pass, ID incorporates in γ' the public states of the nodes in $\mathcal{N}$ that belong to the new nodes in $\mathcal{P}(\mathrm{ID})$, together with the received openings for each such node, and aborts otherwise. Next, if any v in the new $\mathcal{P}(\mathrm{ID})$ set is affected by an update U_a, they overwrite its public key, parent hash value, signature, identifier, transcript hash value, and confirmation tag to the one set by U_a, and update the unmerged leaves and predecessor keys appropriate; and else, if corresponding to a newly populated leaf, determine the corresponding added party from (U, R, A) and add the new public key and identifier ID^* to the leaf. If several nodes in their state are affected by updates, they also check that for every such node in their path, the update setting a new state for it is the same setting a state for one of its parents. Once the updating of the public state of $\mathcal{T}$ is done, they run $\mathrm{verifyRH}(\gamma', M)$, aborting if the output is 0. Once those verifications are passed, for all nodes affected by some U_i, they decrypt the appropriate seed, derive the new key-pairs from it as in algorithm $\mathrm{re\text{-}key}$, check that the received public key matches the derived one, aborting if not, and otherwise, overwrite the public and secret keys with them; set $\mathrm{Unmerged}(v) \leftarrow \emptyset$, and then $\mathrm{Unmerged}(v) \leftarrow \mathrm{Unmerged}(v) \cup \text{ł}_i$ for each leaf l_i that is an ancestor of v corresponding to an added party. After that, they update $\gamma.G$ to account for membership changes as per R and A. Finally, they compute the key schedule for the current round, set $\gamma \leftarrow \gamma'$, deleting both the old key schedule and the old key material from node states, and delete $\gamma' \leftarrow \emptyset$.

[14] Observe that this could allow an active adversary to continuously send inconsistent messages, preventing users from updating. Since this falls outside of our model, we do not consider it here for simplicity, but note that it could be prevented by having users process all operations that do verify and compute an updated round hash, hashing together the received value and the operations that failed verification, inputting this into the transcript hash instead. This would ensure that parties agree on the transcript hash if and only if they processed exactly the same operations.

If the user is not yet part of the group, M will also contain a welcome message $\mathcal{W} = (\mathcal{H}_{round}, \mathcal{H}_{trans}, G, \mathrm{confKey}, \mathrm{initSec})$ together with a copy of the public state of the ratchet tree $\mathcal{T}$, allowing the user to initialize their state prior to executing the instructions above. The newly added user ID_i will first check that G matches the leaf identifiers in $\mathcal{T}$, compute the round hash from $\mathcal{T}$, R and A as in Def. 5.3.1, and check that it matches the received value $\mathcal{H}_{round}$ (and skip this step when later processing the rest of the round message). If any of these checks fails, the user immediately aborts. Next, they will initialize their state γ by setting $\gamma.\mathrm{ID} \leftarrow \mathrm{ID}_i$, $\gamma.\mathcal{H}_{trans} \leftarrow \mathcal{H}_{trans}$, $\gamma.G \leftarrow G$, $\gamma.\mathrm{confKey} \leftarrow \gamma.\mathrm{confKey}$, and $\gamma.\mathrm{initSec} \leftarrow \gamma.\mathrm{initSec}$. Finally, they set the state $\gamma(l)$ of the leaf l to contain the init key with which they were added - note that they will not have at this point knowledge of the secret keys of any other node, but they will obtain some as soon as they process any U_a. When doing so, note that for the verification of the signature they will need to make use of the keys in $\mathcal{T}$. Last, to process an initialization message $I = (\texttt{'init'}, \tilde{\mathrm{ID}}, G, P, S)$, ID verifies the parent hash for the node public states in P, using $\mathrm{PK}_v^{\mathrm{pr}} = 0$ for all nodes $v \in \mathrm{path}_{\tilde{\mathrm{ID}}}$, derives the keys for $\tilde{\mathrm{ID}}$'s path from S, and creates a ratchet tree with users in G as leaves and the obtained keys. Last, they initialize the key schedule, with initial value 0 for $\gamma.\mathrm{initSec}$ and $\mathcal{H}_{trans}$, storing all in the newly created state γ.

Get group key. To extract the current group key a user ID with local state γ fetches $I = \gamma.\mathrm{appSecret}$.

Handling concurrent changes to the group membership. Regarding the considerations on handling concurrent dynamic operations in the collect and deliver operation, note that it is mandatory that removes precede updates, as the new keys sampled by the latter might be encrypted to keys under the knowledge of some of the removed parties; thus, if the node is not blanked, there is not guarantee the removed party will no longer have knowledge of any node's state - our restriction on the ordering prevents this, enforcing that a node is blanked whenever a leaf under it is removed, irrespective of which updates take place. With regards to conflicting removes, i.e., those where the removed party in one is the remover in the other, so that processing the one would render the other syntactically incorrect, this could be left up to the group policy. For example, if two parties ID_i and ID_j concurrently remove each other, different policies could be a) both take effect, b) none take effect and c) only one takes effect. We consider the first option to be the most desirable, e.g. so users could not avoid being removed by issuing removals of other parties, and so apply this one in the protocol.

Protocol type	Rounds to heal t corruptions	Cumulative sender communication		Per-user recipient communication	Subsequent per-user update cost	
		no coordination	coordination		worst	average
(a) corrupted parties unknown						
Original TreeKEM & variants [ACDT20, BBR18, KPPW+21, Mat19]	n	$n^2\log(n)$	$n\log(n)$	$n\log(n)$	$\log(n)$	$\log(n)$
Propose-commit TreeKEM [BBR+23]	2	n^2	n	n	n	n
Bienstock et al. [BDR20]	2	n^2	n	$n^\dagger$	$\log(n)^\dagger$	$\log(n)$
Bidirectional channels [WKHB21a]	2	n^2	n^2	n	n	n
This work	$\lceil\log(n)\rceil+1$	$n\log^2(n)$	$n\log^2(n)$	$\log^2(n)$	$\log(n)$	$\log(n)$
(b) corrupted parties known						
Original TreeKEM & variants [ACDT20, BBR18, KPPW+21, Mat19]	t	$t^2\log(n)$	$t\log(n)$	$t\log(n)$	$\log(n)$	$\log(n)$
Propose-commit TreeKEM [BBR+23]	2	$t^2(1+\log(n/t))$	$t(1+\log(n/t))$	$t(1+\log(n/t))$	$t(1+\log(n/t))$	$\frac{t^2+(n-t)\log(n)}{n}$
Bienstock et al. [BDR20]	2	$t^2(1+\log(n/t))$	$t(1+\log(n/t))$	$t(1+\log(n/t))^\dagger$	$\log(n)^\dagger$	$\log(n)$
Bidirectional channels [WKHB21a]	2	tn	tn	t	n	n
This work	$\lceil\log(n)\rceil+1$	$t\log^2(n)$	$t\log^2(n)$	$\log(n)\cdot\min(t,\log(n))$	$\log(n)$	$\log(n)$

Table 5.5: Comparison of the communication complexity of different CGKA protocols. For a detailed discussion of the table see Section 5.4. The values x depicted in the last 5 columns are to be understood as $\mathcal{O}(x)$. We assume that the ratchet-tree based protocols start with a fully unblanked tree. †: In the uncoordinated case, the protocol's recipient communication is n^2 (case (a)) and $t^2(1+\log(n/t))$ (case (b)), respectively. Regarding the subsequent update cost, while the protocol formally has a worst case subsequent update cost of $\log(n)$, it is only secure in a weak security model. Modifying it to obtain PCS guarantees similar to the other protocols, e.g. by tainting [KPPW+21], would lead to future worst-case update cost of n (case (a)) and $t(1+\log(n/t))$ (case (b)), respectively.

5.4 Efficiency

In this section we discuss the communication complexity of our protocol and compare it with other CGKA schemes. We focus on the cost incurred by several users updating concurrently to recover from compromise, as this is the main setting we aim to tackle with this work. An overview is given in Table 5.5.

Considered setting. Not only does the sequence of operations preceding concurrent update operations (in the case of ratchet-tree based CGKA schemes) have a crucial impact on the resulting communication cost, but also, whether the participating parties know which of the other parties have been compromised and when they are planning to update. Among the different settings one could compare, we restrict our view to the following, quite natural in our opinion.

We consider a group of n users, t of which have been compromised. For ratchet-tree-based protocols we assume that the tree is fully unblanked / untainted, as this should typically be the case, with updates being the most common operation. Our analysis differentiates between the settings (a) where it is only known that the group has been compromised, but not who the particular t corrupted users are, and (b) where the set of compromised users is known to everyone. Note that the former essentially forces every member of the group to update, while in the latter scenario only the t compromised users have to act.

The first value we are interested in is the number of rounds of (potentially) concurrent

updates, after which the group key is guaranteed to be secure again. The second is the cumulative sender complexity (measured over all rounds), which essentially corresponds to the number of public keys and ciphertexts sent to the server. Here, we again distinguish between two settings. Namely, whether the parties act coordinated or not. In the latter case the participating parties are not aware of whether other parties are concurrently preparing updates/commits, which, depending on the scheme, potentially leads to the server having to reject packages. In the former case, on the other hand, they have this knowledge. In practice, this could be implemented by introducing an additional mechanism, that requires parties to wait for a confirmation by the server before preparing and sending update packages. We further track the per-user recipient communication complexity, again measured as a total over all rounds required to recover from compromise. The final considered value is the sender communication cost of a single, non-concurrent, update/commit in a subsequent round. Here, we state both the cost of the worst-case party as well as the average cost.

In Table 5.5 we mark schemes that perform substantially better or worse in one of the categories in green and red, respectively.

The communication complexity of CoCoA. We first discuss the number of rounds required to recover from compromise of t users. As we will show in Section 5.5, it is sufficient for the group to recover that all corrupted users concurrently update in $\lceil \log(n) \rceil + 1$ rounds.

Regarding the sender communication complexity, the size of update packages sent by a user ID to update in the CoCoA protocol is proportional to the size of the resolution of ID's co-path, which will be of order $\log(n)$ for a fully unblanked tree[15]. However, this value could be up to linear in a tree with many blanks, as is the case in TreeKEM and its variants, where blanks (or taints in the case of TTKEM) degrade communication efficiency. In CoCoA concurrent updates are merged and thus none are ever rejected by the server. Hence, in the considered scenario CoCoA in both the coordinated and uncoordinated setting has the same sender communication complexity of order $n \log(n)^2$ (corresponding to n users sending an update of size $\log(n)$ in $\lceil \log(n) \rceil + 1$ many rounds) and $t \log(n)^2$ (corresponding to t users sending an update of size $\log(n)$ in $\lceil \log(n) \rceil + 1$ many rounds), for cases (a) and (b) respectively.

With regards to the recipient communication complexity, user ID in our protocol needs to only receive at most a single ciphertext per update (zero if said update does not rotate the keys of any node in their state), and never more than $\mathrm{path}(\mathrm{ID}) = \lceil \log(n) \rceil$ in total. They will also receive at most $|\mathcal{P}(\mathrm{ID})|$ public keys per round.[16] Thus in case (a)

[15] an additional ciphertext would need to be sent for each unmerged leaf across ID's path, but this will not account for much in typical protocol executions.

[16] Note that the size of $\mathcal{P}(\mathrm{ID})$ grows at most by 1 per every blank node.

ID would incur a download cost of order $\log(n)$ per round, and $\mathcal{O}(\log(n)^2)$ across the $\lceil \log(n) \rceil + 1$ rounds. In case (b) only t parties are updating per round, implying that the per round recipient cost is of order $\min(t, \log(n))$ and the cost over all $\lceil \log(n) \rceil + 1$ rounds is of order $\log(n) \cdot \min(t, \log(n))$. Finally, as in CoCoA concurrent updates do not affect the ratchet tree structure and in particular do not require blanks, the cost of subsequent updates remains of order $\log(n)$.

The communication complexity of other CGKA schemes. We now give a brief overview on the communication cost of other CGKA schemes in the considered scenarios as presented in Table 5.5. The first considered class are ratchet-tree based schemes that do not rely on the propose-commit framework, as (the original non-concurrent) TreeKEM v7 and earlier versions [BBR18], rTreeKEM [ACDT20], TTKEM [KPPW+21], and Causal TreeKEM [Mat19]. These schemes require n (in case (a)) or t (in case (b)) rounds to recover, as only one update per round can be implemented[17]. Regarding the sender communication complexity, in the uncoordinated case every (in case (b) corrupted) user would try to update in every round, thus leading cost of order $n^2 \log(n)$ (case (a)) and $t^2 \log(n)$ (case (b)), respectively. In the coordinated case, only one user per round would send an update attempt, leading to costs of $n \log(n)$ and $t \log(n)$. The per-user recipient cost is of order $n \log(n)$ as n packets of size $\log(n)$ have to be downloaded, and, finally, the cost of subsequent updates is of order $\log(n)$, as updating does not result in blanks.

The second class of protocols are ratchet-tree based protocols following the propose-commit paradigm as TreeKEM v8 [BBR+23] and later versions, and the protocol by Bienstock et al. [BDR20]. In these protocols the group can recover very quickly, by all compromised users proposing an update in the first round, and having someone send a commit to all updates of the previous round in a second round. But this comes at the elevated cost of blanking (or tainting in the case of a PCS version of [BDR20]) the paths of all those T users. This leads to the commit having a cost roughly proportional to the number of updates of the previous round. In case (a) this leads to a cumulative sender communication of n^2 in the uncoordinated case (all n user try to commit at cost n) and, with coordination, of n (only one user commits). In case (b) the authors of [BDR20] show that the cost of a commit to t updates in their protocol is of order $t(1 = \log(n/t))$ that also applies to propose-commit TreeKEM. Thus, in this case the sender complexity is given by $t^2(1 = \log(n/t))$ (no coordination) and $t(1 = \log(n/t))$ (coordination), respectively. The per-user recipient communication is of order n (case (a)) and $t(1 = \log(n/t))$ (case (b)). The ability to recover in only

[17] Causal TreeKEM proposes an interesting idea of re-randomizing node secrets through a concrete homomorphic operation, instead of re-sampling them. Thus it actually allows for concurrent updates. However, the presented security statement still requires updates of every compromised party in *different* rounds, thus leading to communication complexity as presented in the table.

two rounds in propose-commit TreeKEM comes at the cost of introducing blank nodes in the ratchet tree. Concretely, in case (a) the commit to n updates would lead to a fully blanked tree, meaning that the average (and worst case) cost of subsequent updates is going to be linear in n. Note that fully recovering from this state requires linearly many commits to single updates. In case (b) there always exists a user with a subsequent update cost of order $t(1 + \log(n/t))$ and the average update cost is at least of order $(t^2 + (n - t)\log(n))/n$. While the protocol of [BDR20] formally has a worst case subsequent update cost of $\log(n)$, it is only secure in a weak security model. Modifying it to obtain PCS guarantees similar to the other protocols, e.g. by tainting [KPPW+21], would lead to future worst-case update costs matching the ones of propose-commit TreeKEM.

Finally, the protocol by Weidner et al. [WKHB21a] based on bidirectional channels also supports concurrent operations and allows the group to recover in 2 rounds. We point out that this work targets a different network model and has thus a different focus than ours. In particular, the communication complexity for each update is linear in the number of parties, which is considered impractical in the contexts we are interested in. The protocol has a cumulative sender complexity of order n^2 (case (a)) and tn (case (b)) both in the coordinated and uncoordinated case, and a per-user recipient complexity of order n and t, respectively. The cost of subsequent updates is of order n.

Summary and comparison. CoCoA diverges across two different axes from what could be considered a common paradigm until now. On the one hand, users are no longer required to keep track of the full state of the ratchet tree, reducing the recipient communication cost and the storage costs for users, and making this cost differ substantially from the total amount of upload communication. Indeed, this is a big change, as this distinction is not really present in previous works, where the majority of uploaded packets are downloaded by everyone. On the other hand, we consider a more flexible PCS guarantee that only requires users to heal after $\lceil \log(n) \rceil + 1$ rounds. This is in contrast to previous works requiring PCS to hold after a constant number of rounds or only after n rounds. The effect of allowing concurrent updates to be merged is that, on one hand, the protocol is agnostic to coordination, i.e., no additional mechanism is needed that ensures that users do not send update/commit packages that will be rejected by the server, and, on the other hand, it allows the protocol to handle concurrent update operations without introducing blanks in the ratchet tree.

The trade-off with (the non-concurrent) TreeKEM versions that precede the P&C paradigm is clear: we are paying a $\log(n)$ factor in sender communication in exchange for faster PCS that is independent from the number of compromised users. The comparison with P&C TreeKEM is not as straightforward, as the t compromised users can heal in only 2 rounds. The main advantage CoCoA over has this scheme is that it does not introduce blanks in the ratchet tree when handling concurrent operations,

which leads to an improved update cost in subsequent rounds. However, this comes at the cost of slower healing and a factor of $\log^2(n)$ (or roughly $\log(n)$ in case (b)) in sender communication cost. We point out that the P&C framework of TreeKEM allows for more flexibility, e.g. by performing the required updates in several batches over multiple rounds. The exact trade-off achieved by such an intermediate approach is hard to quantify, but, again, due to blanking the cost of future updates will suffer. Finally, CoCoA has the advantage, over all versions of TreeKEM, of reduced recipient communication complexity and that users can prepare updates without the need of extra communication with the server to prevent rejection of said updates.

As a final remark, CoCoA seems to have a slightly worse efficiency than TreeKEM based protocols predating the P&C paradigm, since it requires slightly larger sender communication overall. However, as we show in Section 5.5, this is only the case if fast PCS is required for many users. In fact, a round with a single update will immediately grant PCS to its sender, just as in TreeKEM. Thus, CoCoA can be seen as an extension of pre-P&C TreeKEM, which incorporates the possibility of trading bandwidth for faster collective healing.

5.5 Security

Given a set of parties whose state has leaked, TreeKEM and related variants achieve PCS exactly after all of them perform an update. This is still true in our protocol *as long as the updates are applied sequentially*. Furthermore, we also allow for concurrent updates, which results in some updates only being applied partially. Not surprisingly, it is not sufficient for every party to perform a partial update in order to achieve PCS. Consider the following scenario: parties ID_i and ID_j, both of which were corrupted since their last update, update concurrently by generating and processing simultaneously messages M_i and M_j, respectively, resulting in a round message that refreshes both their paths; and let's say that $M_i \prec M_j$. Then, the seed of the node at the intersection of both paths gets encrypted under a node in the path of ID_j; in particular, it gets encrypted under a key the adversary knows. Thus, PCS is not achieved. However, as a group the two users have made progress towards achieving PCS: all nodes up to that intersection have healed. And hence, if at least one of the parties (potentially concurrently with other users) updates again, the ratchet tree will recover. In our security proof we capture this in more generality. In particular, we show that parties may heal after an individual, non-concurrent update, or by at most logarithmically many updates.

We consider our security proof a significant contribution of this work, not only because it increases our confidence in the security of our protocol, but also because there are significant differences to prior work. There are two main sources of obstacles for us to

overcome: 1) Parties only have partial views of the key material in the tree and are potentially not even aware of changes to the key material outside of their view. This introduces difficulties in defining when a party should consider another party as *safe*. 2) The delivery server is not purely a relay server that one can force to behave honestly using signatures. In our case, the server is expected to actively compute parts of the messages that are exchanged, but still we do not want to put any trust in the server. Accordingly, the adversary gains additional active capabilities compared to prior works of similar flavor. Another aspect new to our proof is a combinatorial result over the challenge graph that establishes the ratchet tree's healing after $\log(n)$ updates.

5.5.1 Security Model and Safe Predicate

To analyze the security of CoCoA, we essentially use the security model from [KPPW$^+$21], introduced in Section 2.4, which allows the adversary to act partially actively and fully adaptively: in this model, the adversary can adaptively decide which users perform which operations, and can actively control the delivery server; however it can not issue messages on behalf of the users. In [KPPW$^+$21] this is enforced by assuming authenticated channels. Since in CoCoA the signing of protocol messages is more involved, parent hash plays an important role also for security against partially active adversaries, and the server no longer just relays messages, we make the use of signatures explicit in this work. As we restrict our analysis to partially active adversaries, the adversary does not get access to signing keys via corruptions. While this might look artificial, it has importance in practice as discussed in Section 1.1.2, and we still obtain meaningful results in the vein of [KPPW$^+$21]. Nevertheless, we consider the analysis of CoCoA's security against fully active adversaries an important question for future work.

Except for explicit signatures, the differences in the setting of *concurrent* CGKA to the one of [KPPW$^+$21] are that 1) users process concurrent messages, 2) no messages are ever rejected by the server, and 3) the server is allowed to send arbitrary (potentially malformed) messages. Regarding 2), it is however possible that messages get lost and even that a user does not process an update they generated. Whether a user ID_i's update message (and which one) is contained in a round message M_i, is represented by a counter c_i. Finally, regarding 3), while our security notion is strictly stronger than the one from [KPPW$^+$21] (where the server could only forward existing messages), the security of protocols such as TreeKEM and TTKEM can trivially be upgraded to our notion: This is true since round messages in these protocols only consist of *signed* messages and the adversary does not learn any party's signing key. In our protocols, in contrast, the server is assumed to perform some computation on users' messages, hence it makes sense to consider a stronger model where this computation is not trusted.

Defintion 5.5.1 (Asynchronous CGKA Security)**.** *The security for CGKA is modeled using a game between a challenger* C *and an adversary* A*. At the beginning of the*

game, the adversary queries $\texttt{create-group}(G)$ *and the challenger initializes the group* G *with identities* $(\mathrm{ID}_1, \ldots, \mathrm{ID}_\ell)$*. The adversary* A *can then make a sequence of queries, enumerated below, in any arbitrary order. On a high level,* `add-user` *and* `remove-user` *allow the adversary to control the structure of the group, whereas* `process` *allows it to control the scheduling of the messages. The query* `update` *simulates the refreshing of a local state. Finally,* `start-corrupt` *and* `end-corrupt` *enable the adversary to corrupt the users for a time period. The entire state and random coins of a corrupted user are leaked to the adversary during this period, except for the user's signing key.*

1. $\texttt{add-user}(\mathrm{ID}, \mathrm{ID}')$*: a user* ID *requests to add another user* ID' *to the group.*
2. $\texttt{remove-user}(\mathrm{ID}, \mathrm{ID}')$*: a user* ID *requests to remove another user* ID' *from the group.*
3. $\texttt{update}(\mathrm{ID})$*: the user* ID *requests to refresh its current local state* γ*.*
4. $\texttt{process}(M, \mathrm{ID})$*: for some message* M *and party* ID*, this action sends* M *to* ID *which immediately processes it.*
5. $\texttt{start-corrupt}(\mathrm{ID})$*: from now on the entire internal state and randomness of* ID *except for the signing key* $\mathrm{ssk}_{\mathrm{ID}}$ *is leaked to the adversary.*
6. $\texttt{end-corrupt}(\mathrm{ID})$*: ends the leakage of user* ID*'s internal state and randomness to the adversary.*
7. $\texttt{challenge}(q^*)$*:* A *picks a query* q^* *corresponding to an action* $\mathrm{a}^* = \texttt{update}(\mathrm{ID})$ *or the initialization (if* $q^* = 0$*). Let* I_0 *denote the group key that is sampled during this operation and* I_1 *be a fresh random key. The challenger tosses a coin* b *and – if the safe predicate below is satisfied – the key* I_b *is given to the adversary (if the predicate is not satisfied the adversary gets nothing).*

At the end of the game, the adversary outputs a bit b' *and wins if* $b' = b$*. We call a CGKA scheme CGKA-secure if for any PPT adversary* A *it holds that*

$$\mathsf{Adv}_{\mathsf{CGKA}}(\mathsf{A}) := |\mathsf{Pr}[1 \leftarrow \mathsf{A} | b = 0] - \mathsf{Pr}[1 \leftarrow \mathsf{A} | b = 1]|$$

is negligible.

In contrast to the security definition of [KPPW+21], process queries do not point to specific queries here. Thus, in order to define our safe predicate, we first need to define what we mean by saying that a party processed another party's update.

Defintion 5.5.2. *Let* ID *and* ID^* *be two (not necessarily different) users and* $(\gamma_q, M) \leftarrow \mathrm{CGKA.Upd}(\gamma_{q-1})$ *an update with associated counter* c*, generated by* ID *in query* q*. Let* $\mathcal{R}(\mathrm{ID}, \gamma_q)$ *be the set of round messages* M *that*

(a) are efficiently computable from the public transcript and private states of all parties,

(b) have counter c *for party* ID*, and*

(c) will be accepted by ID *in state* γ_q*, i.e.,* $\mathrm{CGKA.Proc}(\gamma_q, M)$ *outputs a new state* γ_{q+1} *such that* $\mathrm{CGKA.Key}(\gamma_{q+1}) \neq \mathrm{CGKA.Key}(\gamma_q)$.

Then we say that ID^* processes the update T (or equivalently q) at time $q^* > q$ *if* ID^* *processes some round message* M^* *at time* q^* *resulting in state* γ_{q^*}*, and* $\mathrm{CGKA.Key}(\gamma_{q^*}) \in \{\mathrm{CGKA.Key}(\mathrm{CGKA.Proc}(\gamma_q, M)) \mid M \in \mathcal{R}(\mathrm{ID}, \gamma_q)\}$.

As a special case we say that ID^* *processes the* single *update* T *(or equivalently* q*), if in item (c) additionally the only changes to* $\mathcal{P}(\mathrm{ID})$ *resulting from updates are due to* T.

With this notion in place, we will now define the safe predicate similar to the one in [KPPW+21]. In particular, it rules out all trivial winning strategies, while preserving simplicity by ignoring protocol-specific details such as the relative position of users within the tree.

Defintion 5.5.3 (Critical window, safe user)**.** *Let* ID *and* ID^* *be two (not necessarily different) users and* $q^* \in [Q]_0$ *be some* `update`$(\cdot)$ *or* `create-group`$(\cdot)$ *query. Let* $q^- < q^*$ *be maximal such that one of the following holds:*

- *There exist* $L := \lceil \log(n) \rceil + 1$ *update queries* $\mathrm{a}^i_{\mathrm{ID}} := \mathtt{update}(\mathrm{ID})$ *(*$i \in [L]$*) that were generated for* ID *and processed by* ID^* *within the time interval* $[q^-, q^*]$*. If* ID^* *does not process* L *such queries then we set* $q^- = 1$*, the first query. We denote the last such update query as* q^L.
- *There exists an update query* $\mathrm{a}^-_{\mathrm{ID}} := \mathtt{update}(\mathrm{ID})$ *that was generated by* ID *and processed by* ID^* *as a* single *update within the time interval* $[q^-, q^*]$*. In this case, we set* $q^L := q^-$.

Furthermore, let $q^+ > q^L$ *be the first query that invalidates* ID*'s current key (in the view of* ID^**), i.e., in query* q^+*,* ID *processes a (partial) update* $\mathrm{a}^+_{\mathrm{ID}} := \mathtt{update}(\mathrm{ID}) \notin \{\mathrm{a}^i_{\mathrm{ID}}\}_{i \in [L]}$*. If* ID *does not process any such query then we set* $q^+ = Q$*, the last query. We say that the window* $[q^-, q^+]$ *is* critical *for* ID *at time* q^* *in the view of* ID^**. Moreover, if the user* ID *is* not corrupted *at any time point in the critical window, we say that* ID *is* safe *at time* q^* *in the view of* ID^*.

Similar to [KPPW+21], we define a group key as *safe* if all the users that ID^* considers to be in the group are individually safe, i.e., not corrupted in their critical windows, in the view of ID^*.

Defintion 5.5.4 (Safe predicate)**.** *Let I^* be a group key generated in an action*

$$\mathrm{a}^* \in \{\texttt{update}(\mathrm{ID}^*), \texttt{create-group}(\mathrm{ID}^*, \cdot)\}$$

at time point $q^* \in [Q]_0$ and let G^* be the set of users which would end up in the group if query q^* was processed, as viewed by the generating user ID^*. Then the key I^* is considered safe *if for all users $\mathrm{ID} \in G^*$ (including ID^*) we have that ID is safe at time q^* in the view of ID^* (as per Definition 5.5.3).*

Note that the second case in Definition 5.5.3 exactly captures the case where only *single* updates are accepted in each round. Thus, the security of CoCoA is strictly stronger than sequential variants of TreeKEM. Further, the bound of $\lceil \log(n) \rceil + 1$ updates as required in Definition 5.5.3 is indeed tight, an example is shown in Section 5.5.2, Figure 5.4.

Remark. *We make the following observations about Definition 5.5.4.*

- *If we were to consider a weaker security model where the delivery server is honest, Definitions 5.5.3 and 5.5.4 could be considerably simplified as follows: Let G^* be the group of users at time q^*. Then q^* is safe, if all users $\mathrm{ID} \in G^*$ either did $\lceil \log(n) \rceil + 1$ partial or one full update* before q^**, and one update (partial or full)* after q^**, and are not compromised between/during these updates.*

- *A more permissible safe predicate could be defined when considering the relative position of updating users within the tree. To this aim, one could define a function that evaluates the* progress *a user makes during a concurrent update. This progress corresponds to the number of keys the party updates along its path to the root and depends on the ordering imposed on concurrent updates. e.g. a single update refreshes the entire path, hence makes progress $\lceil \log(n) \rceil + 1$, a concurrent update makes progress at least 1. An even stronger notion of safe group keys could be defined by also considering the effect updates of other users have on a member ID's path: E.g. if ID does a single concurrent update, all keys along her path can heal if her sibling continues to update. In this work, however, we prefer to define a simple and protocol-independent security notion that allows to compare our scheme to other constructions.*

5.5.2 Security of CoCoA

Regarding the security of CoCoA we obtain the following.

Theorem 5.5.5. *If the encryption scheme used in CoCoA is IND-CPA-secure, the signature scheme is UF-CMA-secure, and the used hash functions are modeled as random oracles, then CoCoA is CGKA-secure.*

To prove security of CoCoA, we follow the approach of [KPPW+21] and consider the graph structure that is generated throughout the security experiment. A node i in the so-called *CGKA graph* is associated with seeds Δ_i and $s_i := \mathsf{H}_2(\Delta_i)$, and a key-pair $(pk_i, sk_i) := \mathsf{Gen}(s_i)$. The edges of the graph, on the other hand, are induced by dependencies via the hash function H_1 or (public-key) encryptions. To be more precise, an edge (i, j) corresponds to either:

(a) a ciphertext of the form $\mathsf{Enc}_{\mathrm{pk}_i}(\Delta_j)$; or

(b) an application of H_1 of the form $\Delta_j = \mathsf{H}_1(\Delta_i)$ used in hierarchical derivation.

Naturally, the structure of the CGKA graph depends on the `update`, `add-user` or `remove-user` queries made by the adversary, and is therefore generated adaptively. To argue security of a challenge group key, we consider the subgraph of the CGKA graph that consists of all ancestors of the node associated to the challenge group key – the so-called *challenge graph*. By functionality of the CGKA protocol, the challenge group key can be derived from any secret key/seed associated to a node in the challenge graph. For an example of CGKA graph and challenge graph see Figure 5.4. To argue security, none of the secret keys in the challenge graph must be leaked to the adversary by corruption. We prove (in Lemma 5.5.6 below) that this is indeed the case for CoCoA if the safe predicate is satisfied. Our proof follows the ideas from [KPPW+21], but involves a new combinatorial argument to establish the upper bound of $\lceil \log(n) \rceil + 1$ updates for healing the state of every user. Further, the fact that in CoCoA users only keep track of a part of the ratchet tree substantially complicates the proof of this statement.

In more detail, the proof for the protocol in [KPPW+21] relies on the property that every key in the challenge graph must stem from an update that the party ID^*, who generated the challenge key, processed. This can easily be ensured for protocols keeping track of the full ratchet tree, by forcing parties, who do not agree for every point in time in the protocol execution on every key associated to a node in the ratchet tree, into inconsistent states, thus making future communication between them impossible. Note that this implies the desired property. In this case, if a user ID, while generating an update, encrypts the seed of a key pk to some pk', and later ID^* encrypts to pk,

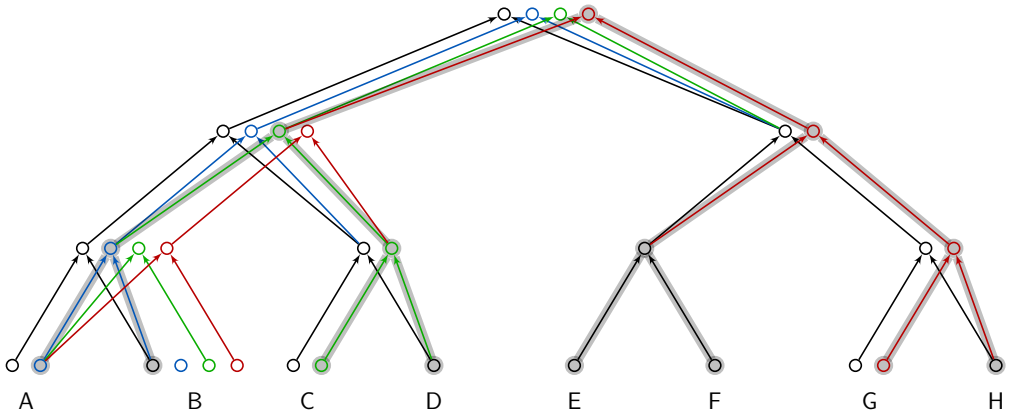

Figure 5.4: Example; CGKA graph and challenge graph. Sequence of operations; we write $\mathrm{update}(X \prec Y)$, to indicate that parties X and Y updated concurrently and X's update took precedence over Y's. A group with 8 parties is set up (black), $\mathrm{update}(A \prec B)$ (blue), $\mathrm{update}(C \prec B)$ (green), $\mathrm{update}(G \prec B)$ (red), G's update is challenged. Vertices and edges that are part of the challenge graph are shaded in gray. Note that even though B updated three times her leaf key in the challenge graph lags behind by $3 = \log(8)$ steps.

then ID^* must have had pk' in their state at some point in time, and thus in particular processed the update establishing it.

Unfortunately, while in an execution where the server behaves honestly, this property would also be true with respect to the relatively simple definition of processing an update of Definition 5.5.2, it is no longer true if we allow an untrusted server. Since in CoCoA the server might send malformed round messages, this property turns out to not hold anymore. We overcome this issue by giving a more involved definition (which is equivalent to Definition 5.5.2 in the honest server setting) of *weakly processing* an update and then essentially show, in the ROM, that every key in a user's state must stem from a weakly processed update (Lemma 5.5.11). Further, we show that users that do not agree on the same history of weakly processed updates transition to inconsistent states (Lemma 5.5.10). For this we have to show, for example, that all keys introduced into a user's state after a change to the resolution of their copath must have been weakly processed in an earlier round (even in the case that at this point in time this update did not affect the user's limited view of the ratchet tree). To prove these properties we rely on the consistency mechanisms of transcript hash and parent hash.

With these statements in place we are finally able to show that no key in the challenge graph is leaked to the adversary, where we use the observation that this property holding with respect to processing an update as defined in Definition 5.5.2 is implied by it holding with respect to the relaxed definition.

5.5.3 Overview of Proof Structure

In this section we give an overview of the proof structure, introducing the definition of *weakly processing* and a few small preliminary results, along with the statements of the main lemmas we build the proof on. For all security statements in the remainder of the paper we assume all hash function H_i, $i \in [5]$, to be random oracles. Our security proof of CoCoA is based on the proof of the following Lemma.

Lemma 5.5.6. *Assume that* Sig *is a secure signature scheme and* H_3 *a hash function. Then, for any* safe *challenge group key in the asynchronous CGKA security game instantiated with CoCoA, it holds that none of the seeds and secret keys in the challenge graph are leaked to the adversary via corruption.*

With this Lemma in place, Theorem 5.5.5 now follows from the corresponding results in [KPPW$^+$21, Section 3.5]. There, security of a variant of TreeKEM called TTKEM is proven by reducing to a game that is known as generalized selective decryption (GSD), where the adversary can query for encryptions of secret keys under other keys, corrupt keys, and finally has to distinguish a key from random that it did not trivially learn by its previous queries. While GSD had only been defined in the secret-key setting, in [KPPW$^+$21] the authors introduce and analyze a public key variant of this game in the random oracle model, which allows to consider the security experiment of ratchet-tree based CGKA protocols as a special case of GSD. This is true thanks to Lemma 5.5.6, which guarantees that any safe challenge in the CGKA security game implies a valid challenge in the GSD game. The security bound for CoCoA now follows analogously to the corresponding result for TTKEM [KPPW$^+$21, Theorem 4] from the more general result on public-key GSD in the ROM [KPPW$^+$21, Theorem 3].

In order to prove Lemma 5.5.6 we will rely on the notion of weakly processing an update. Since an adversarial server can send wrong openings to a user, when ID processes a round message, they might still be introducing into their state some keys or some hash values that depend on updates they have not processed as per the definition above. For example, consider the case of a round message sent to ID, containing a single update from a user ID^* who is not their sibling: a malicious server could include the correct keys and encryptions in the round message, but substitute the new Merkle commitment by an arbitrary value; this will cause ID to still accept the message and update the corresponding keys in their state, while not having processed ID^*'s update (since the server cannot fool the latter into accepting a round message with a round hash dependent on the wrong opening). Below we define a notion of *weakly processing* an update, which captures this scenario. Before we do that, however, we will define the further notion of *explicitly processing*, capturing the process of users introducing keys into their state by means of (weakly) processing an update. Note, however that

explicitly processing an update will not imply processing it, for the above mentioned reason.

In what follows the adversary queries as defined in the security game (Definition 5.5.1) are also denoted using the protocol notation (Definition 2.3.1) in order to explicit refer to the CGKA protocol in use.

Defintion 5.5.7 (Explicitly Process)**.** *We say that a user* ID^* *has explicitly processed the update* U *generated by user* ID *in query* $q = \mathrm{CGKA.Upd}(\gamma_{\mathrm{ID}})$ *if* ID^* *processed and accepted a round message* M *in query* $q^* > q$ *and either*

- M *contained a counter* c *corresponding to update* U *generated by* ID*, or*
- M *contained a tuple* $p_v = (\mathrm{pk}_v, h_v, \mathcal{H}_{trans}, \mathrm{confTag}, \sigma_v)$ *corresponding to the public state of some node* v*, where* $p_v \in U$ *and its transcript hash value* $\mathcal{H}_{trans}$ *is the same as the one in the state of* ID*.*

Defintion 5.5.8. *We say that a user* ID^* *has weakly processed the update* U *generated by user* ID *in query* $q = \mathrm{CGKA.Upd}(\gamma_{\mathrm{ID}})$ *if* ID^* *processed a round message* M^* *in query* $q^* > q$ *and either:*

- q^* *was the first query* ID^* *processed, and the full tree which* ID^* *received in the welcome message contained a node state sampled in* U*,*
- *there exists* ID' *(not necessarily different from* ID^**), in the group in the view of* ID^**, such that the following conditions below hold. Let first* q' *be either* $q' = \mathrm{CGKA.Upd}(\gamma')$ *the last update from* ID' *weakly processed by* ID^* *or, if such update does not exist, the query in which* ID' *processed their welcome message into the group, initializing state* γ'*. Then:*
  - *there exists a series of efficiently computable round messages (given access to the RO queries, the public transcript, and private states of all parties)* $M'_1, \ldots, M'_t$ *such that* ID' *with respect to state* γ' *would process (and accept) all* M'_i *in order, where* γ'_i *is the state resulting from processing and accepting* M'_i *with respect to state* γ'_{i-1} *(and* $\gamma'_0 = \gamma'$*).*
  - ID' *explicitly processed* U *by processing* M'_t*.*
  - $\mathrm{CGKA.Key}(\mathrm{CGKA.Proc}(\gamma^{q^*}_{\mathrm{ID}^*}, M^*)) = \mathrm{CGKA.Key}(\gamma'_t)$*, i.e. they derive the same group key after processing the respective messages,*

A few remarks regarding this definition: first, note that the update query q' in the second case of the definition is required to have been weakly processed by ID^*. In fact, it must also hold that if this case is satisfied, such query was also *processed* by ID^*,

since weakly processing but not processing an update implies the user processing will never have consistent states with the update issuer. Second, note that considering the case where $\mathrm{ID}' = \mathrm{ID}^*$ is equivalent to saying ID^* explicitly processed U. In particular, explicitly processing an update means weakly processing it. Similarly, by considering the case where $\mathrm{ID}' = \mathrm{ID}$, we can see that processing also implies weakly processing. We recall again that explicitly processing does not imply processing. From this definition, we can define the predicate $\mathrm{wProcess}(\mathrm{ID}, U, q)$, which evaluates to 1 if ID weakly processed U at time q; and else evaluates to 0.

Below we define formally what it means for a user to join the group at the same time or after another user. Note that this is defined with respect to the views that users have of the group. With regards to the situation where a user leaves the group and joins again, we assume that it does so with a new identifier. That is, party identifiers are assumed unique and cannot rejoin once removed. Further, for security we assume also that init keys are only used once.

Defintion 5.5.9. *We say that user* ID^* *is added at the same time or after* ID *(in the view of* $\mathrm{ID}*) if there exist* $q_A = \mathrm{CGKA.Add}(\cdot, \mathrm{ID})$ *and* $q_A^* = \mathrm{CGKA.Add}(\cdot, \mathrm{ID}^*)$ *and queries* q *and* q^* *such that* $\gamma_{\mathrm{ID}}^{q} \equiv \gamma_{\mathrm{ID}^*}^{q^*}$ *and, moreover, either* ID *processed* q_A^* *in some query* $\leq q$*, or the equivalence holds for* q *and* q^* *corresponding to the first process queries for each party.*

Throughout the following, for simplicity of notation, we assume that the state of a party is the same before and after issuing an add, remove and update operation, i.e. if q is a query of the form $\mathrm{CGKA.Add}(\mathrm{ID}, \cdot)$, $\mathrm{CGKA.Rem}(\mathrm{ID}, \cdot)$ or $\mathrm{CGKA.Upd}(\gamma_{\mathrm{ID}}^{q-1})$, we consider the states of party ID at $q-1$ and q the same. While in practice this is not the case, since issuing any of these operations will change the pending state γ'_{ID}, which is a part of the local state, it will not change any value in the rest of the state, and thus will not influence any statement regarding states consistency.

When a user ID is added to the group, we count them as effectively processing two round messages, the first one corresponding to the welcome message, the second corresponding to the round message that comes with it. Thus, we will say ID^* has a state consistent with that of ID at the time the latter joined the group if the state of ID^* is consistent with the contents of the welcome message or, equivalently, with the user that added ID. Note that although the welcome message contains no group key $\gamma.\mathrm{appSecret}$, it does contain a transcript hash value $\mathcal{H}_{trans}$, and users having the same transcript hash value in their state is equivalent (up to collision resistance of the random oracle) to users having the same group key, by Proposition 2.

Before beginning with the proofs, we present here an observation that will be used throughout most of them. Informally, this is the following: due to updates being signed and the adversary not having access to signing keys, every key in a user's state and, in

turn, in the challenge tree, must stem from an update or add query. More precisely, any key stemming from an add query, that is introduced in the state, gets delivered in a packet that comes with a signature by the party who authored the add, together with a signature of the party whom that leaf belongs to; and any key introduced in the state by either explicitly processing an update, or by receiving a node state from the server after a removal, is received as part of a tuple $(\mathrm{pk}_v, h_v, \mathcal{H}_{trans}, \mathrm{confTag}, \sigma_v)$, together with a party identifier ID_v. Before accepting, a user will check that σ_v is a signature for $(\mathrm{pk}_v, h_v, \mathrm{PK}_v^{\mathrm{pr}}, \mathcal{H}_{trans}, \mathrm{confTag})$ under ID_v's verification key. Thus, any key, and moreover, any of those signed values in the state of any $v \in \mathcal{P}(\mathrm{ID})$, must have come from an update by ID_v. In other words, if some state in $\mathcal{P}(\mathrm{ID})$ contained a key, or value from the ones above, not corresponding to a query, we could build a forger for the signature scheme with a loss of n in the advantage by guessing the user that supposedly generated the key in question. Note that every update, add, or remove query requires at most $n + 2\log(n)$ signatures (this happens, e.g., in the case of a fully blanked tree).

The following small proposition shows that, as we would expect of the more general definition of weak processing, users can only weakly process updates issued by parties with a consistent state.

Proposition 3. *Let U be an output from $q^* = \mathrm{CGKA.Upd}(\gamma_{\mathrm{ID}^*}^{q^*-1})$, for some user ID^* with state $\gamma_{\mathrm{ID}^*}^{q^*-1}$ at time $q^* - 1$. Let ID be a party with state γ_{ID} and such that $\mathrm{wProcess}(\mathrm{ID}, U, q) = 1$ for some query q, and such that ID was already part of the group before q. Then the states of ID at time $q - 1$ and ID^* at time q^* are consistent.*

Proof. If ID weakly processed U after joining the group, then, by definition there must exist a user ID' and some message M' such that ID' would explicitly process U by processing M' in query q' and such that its state right after would be consistent with the state of ID at q. Now, for ID' to explicitly process U, their state just before processing M', at $q' - 1$, must have been consistent with the state of ID^* at the time they issued U, i.e. at q^*. This is the case since for ID' to accept M', the signed packet contained in said round message and which corresponded to U must have contained the same transcript hash value as ID' had at $q' - 1$ that moment in their state. However, by collision resistance of the random oracle H_3, since the states of ID at q and ID' at q' were consistent, so were their respective states at $q - 1$ and $q' - 1$. But this implies that ID's state at $q - 1$ was consistent with that of ID^* at q^*. □

It follows from the proposition above that no user can weakly process the same update twice:

Corollary 5.5.9.1. *Let $\mathrm{wProcess}(\mathrm{ID}, U, q) = 1$ for some ID, U and q. Then, except with negligible probability, for any $q' \neq q$, $\mathrm{wProcess}(\mathrm{ID}, U, q') = 0$.*

Proof. Assume for contradiction that there is an update U and distinct queries q, q' with $\mathrm{wProcess}(\mathrm{ID}, U, q) = 1$ and $\mathrm{wProcess}(\mathrm{ID}, U, q') = 1$. Let $q < q'$, and assume first that ID did not join the group in q. By Proposition 3, the state of ID at both $q - 1$ and $q' - 1$ must be consistent with the state of U's author at the time U was generated. But this is not possible, since ID has processed and accepted some round message (and therefore updated its state) in between these times.

Suppose now that ID did join the group in q. Since the welcome packet that they received was (except with negligible probability) part of the output of a query, the transcript hash $\mathcal{H}_{trans}$ in it was computed with respect to the round hash value $\mathcal{H}_{round}$ in it. Further, we know $\mathcal{H}_{round}$ is consistent with the tree that ID received upon joining, which contained some node state from U, by the assumption that ID weakly processed U at q. However, this node state from U also contained a transcript hash value. By Proposition 3, ID at $q' - 1$ was consistent with U's author at the time they issued U. In particular, the transcript hash value in U must be the same as the transcript hash value in ID's state at $q' - 1 \geq q$, by Proposition 2. But this is a contradiction to the properties of the random oracle H_3, since the transcript hash value of ID at this time depended on that of ID at q, which itself depended on the transcript hash value in U. □

With this in place, we will be ready to prove Lemma 5.5.6 above, with the help of two further Lemmas, capturing the consistency guarantees given by our round hash and parent hash mechanisms.

The first one states that users with consistent states have weakly processed equivalent sequences of operations.

Lemma 5.5.10. *Let* ID *and* ID^* *be two users with consistent states at time* q*, and such that* ID^* *joined the group at the same time or after* ID*. Then:*

$$\{U \,:\, \exists q' \leq q \text{ s.t. } \mathrm{wProcess}(\mathrm{ID}^*, U, q') = 1\} \subseteq \{U \,:\, \exists q' \leq q \text{ s.t. } \mathrm{wProcess}(\mathrm{ID}, U, q') = 1\}$$

and, for any q_1^*, q_2^**, with* $q_1^* < q_2^*$ *and any updates* U, U' *such that* $\mathrm{wProcess}(\mathrm{ID}^*, U, q_1^*) = \mathrm{wProcess}(\mathrm{ID}^*, U', q_2^*) = 1$*, there exist* q_1 *and* q_2*, with* $q_1 < q_2$ *such that* $\mathrm{wProcess}(\mathrm{ID}, U, q_1) = \mathrm{wProcess}(\mathrm{ID}, U', q_2) = 1$*.*

The second one states that all the keys that a user incorporates into their state as part of the protocol execution stem from adds or updates that they have weakly processed, and moreover, that these do not correspond to operations that have already been superseded by other weakly processed updates.

Lemma 5.5.11. *For any user* ID *and any key* pk_v *in their local state* γ *associated to a node* $v \in \mathcal{P}(\mathrm{ID})$ *at time* q^**, the following hold:*

1. *there is a query $q \in \{1, \ldots, q^*\}$, such that pk_v is either the InitKey of a party whose add operation was processed by ID in query q, or was sampled in some update U that the user had weakly processed in q; i.e., $\mathrm{wProcess}(\mathrm{ID}, U, q) = 1$;*

2. *for any update U' generated before q^*, which affects v and such that $\mathrm{wProcess}(\mathrm{ID}, U', q') = 1$ for some $q' \leq q^*$, it holds that $q' \leq q$.*

To conclude the security proof, in Section 5.5.4 we give the proof of Lemma 5.5.6 assuming Lemmas 5.5.10 and 5.5.11, which are proven afterwards, in Section 5.5.5.

5.5.4 Proof of Lemma 5.5.6

We now prove Lemma 5.5.6. To do so, we first argue that all parties associated to leaves in the challenge graph belong to the group as viewed by the party generating the challenge key.

Lemma 5.5.12. *Let ID^* be the party who generated the challenge key I^* in query q^*. Then, all parties associated to leaves in the challenge graph belong to G^*, the set of group members at time q^* in ID^*'s view.*

Proof of Lemma 5.5.12. Let $\mathrm{ID} \notin G^*$ and assume for contradiction that ID is associated to a leaf in the challenge graph. Let $v_1, \ldots, v_k$ be the path in the challenge path from ID's leaf v_1 to the root v_k. Further, let $i_1 < \cdots < i_\ell$ be such that the edges from (v_{i_j-1}, v_{i_j}) correspond to encryptions, while the other edges on the path stem from hierarchical derivation. Note that either ID^* never considered ID to be in the group, or ID^* processed a `remove-user`$(\cdot, \mathrm{ID})$ message (and no subsequent `add-user`$(\cdot, \mathrm{ID})$ message). Further, it is not possible that $\ell = 1$ since in this case ID^* would have encrypted a seed to the key of v_{i_1-1} that in this case was generated by ID. However, in the course of processing any round message, before incorporating any key into their local state ID^* checks that it comes with a signature which verifies under the key of some user from the corresponding sub-tree, in particular from a user in G^*.

Thus, we may assume $\ell \geq 2$. We denote the user that generated the key at v_{i_j} by ID_j and the corresponding update query by q_j. So, when ID_1 generated the key at v_{i_1} they encrypted its seed to the a key that was generated by ID, implying that ID_1 considered ID to be in the group at this point in time. Further, user ID_2 encrypted the seed of node v_{i_2} to a key generated in update q_1 implying in particular that at time q_2 they had weakly processed the same operations (and in particular the same add/remove operations) as ID_1 at q_1 by Proposition 3 and Lemma 5.5.10, plus potentially some others, but none affecting nodes on the the subtree under v_{i_2-1} (since any such operation would overwrite the keys created in q_1). In particular, they cannot have processed any

$\texttt{remove-user}(\cdot, \mathrm{ID})$ between q_1 and q_2, so they must consider ID to be part of the group.

By repeatedly applying this argument we obtain that $\mathrm{ID}_{\ell-1}$, when generating update $q_{\ell-1}$, considered ID to be part of the group. Similarly, as $\mathrm{ID}^* = \mathrm{ID}_\ell$ processed this update, they also must have considered ID to be in the group at this point in time and, further, cannot have processed a $\texttt{remove-user}(\cdot, \mathrm{ID})$ message before generating the root of the challenge graph at q_ℓ, as, again, otherwise the key at $v_{i_\ell - 1}$ would have been blanked or overwritten; a contradiction.

□

Proof of Lemma 5.5.6. Let ID^* be the party who generated the challenge key I^* in query q^*. Since I^* is safe, for all parties ID in G^*, the set of group members at time q^* in ID^*'s view, ID must be safe in the view of ID^*. We will show that any key in the challenge graph can only be in the state of ID during the critical window $[q^-_{\mathrm{ID}}, q^+_{\mathrm{ID}}]$ of ID in the view of ID^*. Together with Lemma 5.5.12 this will show the result. We note that all the secret keys in the state of ID during its critical window are overwritten in the state of ID *before* the next corruption of ID. This is the case because, by definition of the safe predicate, after ID generated the last update $\mathrm{a}^{\mathsf{L}}_{\mathrm{ID}}$ they (partially) process some further update $\mathrm{a}^+_{\mathrm{ID}}$ before the next corruption. Even though her own update $\mathrm{a}^+_{\mathrm{ID}}$ might only be partially processed, the way updates are merged in $\mathrm{CGKA.Dlv}$ guarantees that *all* keys along ID's path will be refreshed. Moreover, by Corollary 5.5.9.1, ID will not add to their state any keys that were already part of it. It is sufficient to prove that any key in the challenge graph was generated by a party that had already entered its critical window in the view of ID^* when generating the update resulting in this key. This is because seeds can only be encrypted to keys that have already been generated – since every key in the challenge graph must stem from a query, as argued above –, which means that all receiving parties are already in their respective critical window. Accordingly, we now prove that each key in the challenge graph was generated by some user $\mathrm{ID} \in G^*$ within the first period $[q^-_{\mathrm{ID}}, q^L_{\mathrm{ID}}]$ of ID's critical window for q^* in the view of ID^*.

To this aim, first note that the challenge node was generated by user ID^* in query q^*, and q^* by definition lies in the first period of the critical window of ID^*. We will proceed by induction: consider an arbitrary internal node v in the challenge graph, and assume it was generated at time q_v (which is in the first period of the critical window of the generator). Then, all except one parent of v must have been generated *before* query q_v. More precisely, for all parents w of v in the challenge graph it holds: Either (1) w was generated in the same query q_v (this happens for exactly one parent), or (2) w was generated by a party $\mathrm{ID} \in G^*$ in the subtree rooted at w, by Lemma 5.5.12, at some time $q_w < q_v$. In case (1), if v was generated during the first period of the

critical window of its generator, then clearly the same holds for w. In case (2), since ID^* only processes messages from parties with a consistent state, and q_w was weakly processed by the party who generated q_v *before* it generated q_v (by Lemma 5.5.11), having weakly processed q_v, user ID^* must have also weakly processed q_w before. This follows from Lemma 5.5.10, as both users must have then weakly processed the same updates. In other words, we have $q_w^+ < q_v^+$, where q^+ refers to the query when ID^* weakly processes q. Similarly, we can deduce that q_w must have been the last update generated by ID before q_v, which was weakly processed by ID^*. This means that, for all update queries $q'_{\mathrm{ID}} > q_w$ generated by ID that are weakly processed by ID^*, we have $(q'_{\mathrm{ID}})^+ \geq q_v^+$.

We will now look at the critical window of ID. Consider first the case where ID's critical window is defined through a single update q^-_{ID} that was processed individually by ID^*. We will show that $q^-_{\mathrm{ID}} \leq q_w$ (so that, in fact, equality holds here), which means that ID generated q_w in the first period of its critical window. Assume for contradiction that $q^-_{\mathrm{ID}} > q_w$, so that $q^* > (q^-_{\mathrm{ID}})^+ > q_v^+$. Consider the partition of the path in the challenge graph from v to I^* given by encryption edges, as in the proof above of Lemma 5.5.12: let $v_1, \ldots, v_k$ be the nodes in such path, and let $i_1 < \cdots < i_\ell$ be such that the edges (v_{i_j-1}, v_{i_j}) correspond to encryptions, while the other edges correspond to hierarchical derivations. If $\ell = 1$ then, by Lemma 5.5.11, in q^* ID^* encrypted to a key generated in q^-_{ID}, but q^-_{ID} overwrote all keys generated by q_v, so v cannot be in the challenge graph, which contradicts our assumption on v. Assume, thus $\ell \geq 2$ and denote the user that generated the key at v_{i_j} by ID_j and the corresponding update query by q_j. In that case, by the same argument as in the previous case, ID_1 must have not weakly processed q^-_{ID}, before q_v, since otherwise they would encrypt v_{i_1} under a key sampled by the former, and not the latter, and v would not be in the challenge graph. But we can recursively apply this argument to deduce that $\mathrm{ID}_\ell = \mathrm{ID}^*$ must also not have weakly processed q^-_{ID} at q^*, a contradiction.

Finally, consider the case where ID's critical window is defined through a sequence of $L = \lceil \log(n) \rceil + 1$ updates $q^-_{\mathrm{ID}} = q_1, \ldots, q_L$. We will argue with respect to the potentially larger sequence $q^-_{\mathrm{ID}} = q_1, \ldots, q_{L'}$ of weakly processed updates in the first period of the critical window; here $L' \geq L = \lceil \log(n) \rceil + 1$, since processing implies weakly processing. Applying the same argument as in the previous paragraph, we can deduce that at most one of these updates q_i could have been weakly processed by ID^* in the window $[q_w^+ + 1, q_v^+]$, and if this is the case, then this update must have been weakly processed *concurrently* with q_v. Note that if $q_i^+ = q_v^+$ for any $i > 1$, we have $q_1^+ \leq q_w^+$, which by Lemma 5.5.10 implies $q_1 \leq q_w$ and thus ID is in its critical window when it generates q_w. If this is not the case, no q_i can belong to the mentioned interval, meaning it suffices to show that $q_1^+ < q_v^+$, as this implies $q_1 \leq q_w$. Assume for contradiction $q_1^+ \geq q_v^+$. Consider v's child u_1; following the same argument, at most one of ID's queries can be weakly processed in the time interval $[q_v^+ + 1, q_{u_1}^+]$. But

there are at most $\lceil \log(n) \rceil - 1$ descendants of (non-leaf node) v, let these be denoted by $u_1, \ldots, u_l$ with $l \in [\lceil \log(n) \rceil - 1]$ and $u_0 := v$. Now, by the above, for each time interval $[q^+_{u_{i-1}} + 1, q^+_{u_i}]$ ($i \in [\lceil \log(n) \rceil - 1]$) there is at most one (concurrent) update from ID weakly processed by ID^*. Since u_l is the root of the challenge graph, which was created in query q^*, and by assumption $q_1^+ \geq q_v^+$, a simple counting argument implies that $(q_{L'})^+ > q^*$, a contradiction.

□

5.5.5 Proofs of Lemmas 5.5.10 and 5.5.11

At the core of the proofs of both of the lemmas in the previous section (Lemmas 5.5.6 and 5.5.12) lies the fact that an adversary cannot commit to subtrees outside the users' states containing wrong information in a way that will not push users out of consistent states. This is captured by the following game and lemma:

Defintion 5.5.13. *The ability of the adversary to open wrong commitments to the parts of the ratchet tree the users do not see is modeled through the following* OPEN *game, played by an PPT adversary* A*. The game is syntactically equivalent to the CGKA security game from Definition 5.5.1, except for the challenge queries, which now take a different shape, and the winning condition. In this game,* A *is able to issue a query* $\mathrm{OPEN.Chall}(q)$ *at any point, with* q *being a query of the form* $q = \mathrm{CGKA.Proc}(\mathrm{ID}, \cdot)$ *already made in the CGKA game, and output a ratchet tree* T_q*.*

The game outputs $1 \leftarrow$ OPEN *if, after such a* $\mathrm{OPEN.Chall}(q)$ *query, the following is true for* T_q*:*

- T_q *is a well-formed ratchet tree, every node except for its leaves has indegree two, it has no blank leaves and it contains at least one ratchet tree leaf (i.e., one containing a public verification key);*
- $\mathrm{PHash.Ver}(T_q) = 1$*, i.e. parent hash verifies;*
- *the state of every node* v *in* T_q *corresponds to that output by some add or update query* $q_v < q$*;*
- *the label* $\ell(v^{T_q}_{root})$ *(as in Def. 5.3.1) of the node at the root of* T_q *is the same as the label of some node in* $\mathcal{P}^q(\mathrm{ID})$*;*
- *there exist a query* q_i *to which some node state in* T_q *correspond to, that has not been weakly processed by* ID *at time* q *or before.*

Else, $0 \leftarrow$ OPEN*. Finally, we say that* A *wins the OPEN game if* $1 \leftarrow$ OPEN*.*

We now show that no adversary can win game OPEN with more than negligible probability.

Lemma 5.5.14. *The probability that any adversary* A *making at most* Q *queries in the CGKA security game wins the OPEN game with respect to any* $q \in [Q]$ *is negligible.*

In order to prove this, we first prove the existence of an extractor algorithm that can return the queries (weakly) processed by any user at any point in the game. This is formalized in the following lemma.

Lemma 5.5.15. *There exists an efficient algorithm* E *that, given access to the random oracle queries, the public transcript, and private states of all parties, can compute the set of operations* Q *that any user* ID *has weakly processed in any query* $q = \mathrm{CGKA.Proc}(\gamma_{\mathrm{ID}}, M)$.

Proof. Note first that, having access to the private states of the party, E can learn if ID accepts M, just by running $\mathrm{CGKA.Proc}$. Also, it is clear from M which adds and removes are processed, as well as any update that is explicitly processed. We will prove the statement of the lemma by induction, showing that E can compute the set of operations weakly but not explicitly processed by ID, as well as the set of users in the group with whom ID could still have consistent states with.

To begin, note that if M is the first message ID processes, either because they set up the group, or are added to it, E can easily extract all the updates weakly processed by ID. In the first case, these are none, and in the second they are given by the ratchet tree ID receives as part of the welcome message, as by the first bullet point of Definition 5.5.8. Before continuing, we argue that in the latter case these are all updates ID weakly processes, i.e. any update satisfying the second bullet point of said definition must be one of those given by the ratchet tree ID receives. Suppose that was not the case, and that there was some user ID' whom, after processing some round message and, through it, explicitly processed some update U, was in a consistent state with ID. From the fact the states were consistent, we know the respective round hash values for both parties must be the same. However, that of ID' was computed using as input some key sampled in U which, by assumption, was not used in the computation of ID's round hash value. This is however, a contradiction to the collision resistance of the RO H_3.

Now, let U^i_{ID} be the set of users with whom ID could still have consistent states with respect to their state γ^i_{ID}; i.e. those for which a sequence of round messages as described in the second case of Definition 5.5.8 exists. More in detail, if $\mathrm{ID}' \in U^i_{\mathrm{ID}}$, there is a sequence of round messages $M'_1, \ldots, M'_{r_i}$ that ID' would process with respect to their state at query q' (recall this corresponds to either the last update query issued by id'

and weakly processed by ID or the time ID' joined the group), and which would bring them to a state consistent with γ^i_{ID}. We will show how E can compute the sets U^i_{ID} along with the queries weakly processed by ID. The initial set U^i_{ID} can be determined by E as follows. If ID sets up the group, this set starts empty; whereas if ID is added to the group by ID' through an add query $\mathrm{CGKA.Add}$ sampled in query q', this set is $U^1_{\mathrm{ID}} = U^{q'}_{\mathrm{ID}'}$. Further, E can track the impact of adds and removes in these sets as follows. Whenever ID processes and accepts a round message in query $\tilde{q}$ containing a remove or an add, E respectively removes the corresponding party from $U^{\tilde{q}}_{\mathrm{ID}}$, and adds it if it was added by a party that is in $U^{\tilde{q}-1}_{\mathrm{ID}}$. To justify the latter, observe that a newly added party will be added with a state consistent with that of the party who added them.

It remains to show that E, given knowledge of the updates weakly processed by ID up until that point, and the set U^{q-1}_{ID} of parties with whom ID has not been pushed to inconsistent states with, can extract the updates that are weakly processed by ID when processing M and compute U^q_{ID}. Let γ^q be the state of ID after processing M, and let γ^{q-1} be its state before it. In particular, we want to identify the set Q' of updates that were weakly processed but not explicitly processed by ID when transitioning from γ^{q-1} to γ^q.

We start with some terminology and a couple of observations. Given a pair of openings $o_v = (o_{v,1}, o_{v,2})$ associated to node v, received in M during query process q (and supposed to be commitments to the keys on the subtree under v's parents), we say that o_v is a *correct opening pair* if there is are queries to the RO with output $o_{v,1}$ and $o_{v,2}$ and input a tuple $(o_{v_i,1}, o_{v_i,2}, \mathrm{pk}_{v_i})$, for $i \in \{1,2\}$ where $o_{v_i,j}$ are hash values in the image of the RO, and pk_i is either equal to $\mathtt{blank}$, or is the public key associated to the corresponding parent of v in the output of some update query q' issued by user ID' and taking place before q. In the former case, we require that $o_{v_i,1}$ and $o_{v_i,2}$ are also correct openings, or their parents, should the corresponding key be $\mathtt{blank}$, and so on. Moreover, we require that, either the update pk_i to which belongs to was (one of, if several were concurrent) the last affecting that node which ID weakly processed, or that the state of ID' at time q' is consistent with that of ID at $q-1$. This will ensure that if it corresponds to an update not yet weakly processed by ID, this it will be possible for ID to weakly process it at q. Last, we further require that one of the pk_{v_i} belongs to the same update as the pk_v. This, is also needed to ensure ID would process any such update, as any user for which those two nodes are in their state will not accept a round message not satisfying that.

Now, we observe that, for ID to weakly process in q the query to which pk corresponds to or, in fact, any update query coming from a user in v's subtree other than that setting the new key for v, it must be that the opening pair o_v is a correct opening pair. Indeed, for ID to weakly process such update query, there must exist a user ID^* in

the subtree under v that would, with respect to some of its states and a certain chain of round messages, ultimately explicitly process said update, and end up in a state consistent with γ^q. However, said user's state will necessarily include a key for (the resolution of) the parents of v, as well as two openings corresponding to the node's parents (of the nodes in the resolution). Thus, were these openings not correct, either because they were never output by the RO, or because their inputs were not of the right format, or sampled by a user with an inconsistent state, we would have a contradiction to either the preimage or collision resistance of the RO, or the fact that said user would explicitly process the update, respectively. Also, note that that user, before processing that last message in the chain, would have a state consistent with that of ID at $q-1$, and would thus accept said update.

Further, observe that it must be the case that, for any update coming from a user in v's subtree to be weakly (and only weakly) processed by ID, there must be a path of correct openings all the way from v to a leaf. Indeed, ID^*'s state will contain public keys (and implicitly openings) for each of the nodes in its path. Were the above not true, the fact that ID and ID^* end up sharing the same round hash would again imply a contradiction with the properties of the RO.

In order to identify the exclusively weakly processed updates and update U^q_{ID}, E performs the following steps for each of the non-leaf nodes $v \in \mathrm{Res}^q(\text{co-path}(\mathrm{ID}))$, i.e. in the resolution of ID's co-path after processing M, for which its opening pair is correct, following the observations above.

We distinguish several cases. Case (1), $v \in \mathrm{Res}^{q-1}(\text{co-path}(\mathrm{ID}))$, i.e. v is not a node introduced in ID's state in q. E now looks at each of the two openings $o_{v,1}$ and $o_{v,2}$ and distinguishes two further situations in each case, depending on whether the opening values are new. Case (1.1) $o^q_{v,i} = o^{q-1}_{v,i}$, i.e. the opening $o_{v,i}$ does not change by processing M, and (1.2) the opposite, $o^q_{v,i} \neq o^{q-1}_{v,i}$. In case (1.1), E does not add any update to Q' and stops further examining any values or nodes in the subtree under the corresponding parent of v. The reason being that in this case it is not possible for ID to have weakly processed any update coming from a user in said subtree. To see why this is the case, assume for contradiction that it is not. Then there exists some user under said subtree that could explicitly process some update and end up in a state consistent with ID, making ID to have weakly processed it. This user must have been in consistent states with ID before processing the update, and in particular have the same transcript hash value as ID at that time. However, when processing the message through which they explicitly processed this update, the labels for v must necessarily change for them, i.e. the labels for v they input into their computation of the round hash must be different for this process query and the previous one. But then it is impossible for the two of them to have the same round hash value in both rounds and therefore impossible for them to have consistent states, by the collision resistance

of random oracle H_3.

In case (1.2), E recovers the inputs $(o_{w_i,1}, o_{w_i,2}, \mathrm{pk}_i)$ of the RO query with output $o_{v,i}$ (these exist and have this form by the assumption on $o_{v,i}$ belonging to a correct opening pair), where w_i is the corresponding effective parent of v. Then it checks that $(o_{w_i,1}, o_{w_i,2})$ is a correct opening pair, ignoring the subtree under w_i if not, and makes the same distinction as before, now applied on the $o_{w_i,j}$, iteratively excluding subtrees under any node for which its opening pair is not correct, or for which its key corresponds to updates already weakly processed by ID. At the end of this iterative process, E compiles a list of all the updates given by the keys in the correct openings, which had not yet been weakly processed by ID. It remains to determine for which of these updates there are users in U_{ID}^{q-1} which could explicitly process them through processing some round message and arrive at a state consistent with γ^q. To this end, first note that, following the observation above, only users for which there is a path of correct opening pairs from their path to v will be able to process some message and end up in such a consistent state. Thus, E considers the users in U_{ID}^{q-1} for which such a path exists, and looks at the nodes in this path and the resolution of its copath which contained keys (those the openings were committing to) belonging to updates in the previously mentioned list. The updates from this list for which this is the case are then added to Q'. To see why, note that if ID^* is such a user, the message M^* output by collecting all the operations and updates processed or explicitly processed by ID in M, together with those in Q', with the same ordering applied, will be accepted by ID^* (with respect to their state after processing the corresponding messages $M_1^*, \ldots, M_{r_{q-1}}^*$ given by the fact $\mathrm{ID}^* \in U_{\mathrm{ID}}^{q-1}$). Moreover, by setting the openings in M^* corresponding to the parents of nodes in the resolution of ID^*'s copath, to be the same as those given by the RO queries, we ensure that ID^* will derive a state consistent with γ^q. Last, E removes from U_{ID}^q those user for which such path of correct opening pairs from their leaf to v does not exist.

We now examine case 2), where v has been added to ID's state as a result of processing M. In particular, this means that v is in the resolution of a node blanked by some removal in M. Recall that ID receives from the server the states of nodes in a subtree spanning the paths and (resolutions of) co-paths of removed members. A subtree that, in particular, contains a state from v. E first checks whether the key at v after processing M matches that in the subtree (this will not be the case if there was some update concurrent with the removal included in the round message), and adds no query to Q' if this is the case. If it is not, then whichever update the new key at v corresponds to was explicitly processed by ID at q, and E then performs the same steps as in the previous case, looking at the keys commited in the openings if these are correct, and working its way down the subtree under v through examining the RO queries, all the way to the ratchet tree leaves. Note that the removals taking place in M would also be

processed by whichever users under v's subtree could still remain in consistent states with ID, since these will receive node states for the same nodes in the resolutions of all blanked descendants of v (and any openings sent to ID can also be mirrored in the last round message M'_t sent to said user). □

Proof of Lemma 5.5.14. Now, in order to prove the statement of the Lemma, we will argue for contradiction, assuming an adversary A that wins the OPEN game exists. We will show that if A had a non-negligible probability of winning the game at some given query, then it must have had it also with respect to an earlier query. Since there is no chance for A to win the game with respect to the first process query (as this must correspond to either the group creator processing their own update or a user joining the group right after initialization), this will show that all queries corresponding to node states in T_q must have been processed by ID at time or before q, giving us the desired contradiction. Accordingly, consider an execution of the OPEN game, and let q be the first query of the form $q = \mathrm{CGKA.Proc}(\mathrm{ID}, \cdot)$ where A could win, i.e. the first query for which a tree T_q, computable in polynomial time by A and satisfying all conditions in Def. 5.5.13 exists.

By assumption, there is a node $v \in \mathcal{P}^q(\mathrm{ID})$ in the state of ID at time q, with label ℓ_v, which is the same as the label of the root node of T_q. Recall that if the state of v contains openings $o_v = (o_{v,1}, o_{v,2})$ and a public key pk_v, then $\ell_v = \mathsf{H}_3(o^1_v, o^2_v, {}^p\gamma(v))$. Note that we could assume w.l.o.g. that $v \in \mathrm{Res}(\text{co-path}(\mathrm{ID}))$. Indeed, note that for any node in $T_q \bigcap \mathcal{P}^q(\mathrm{ID})$, by collision resistance of H_3, the states of that node in T_q and $\mathcal{P}^q(\mathrm{ID})$ must match. Moreover, ID must have weakly processed (in fact, explicitly processed) any query setting any keys for nodes in $\mathrm{path}(\mathrm{ID})$. Thus if T_q satisfies the conditions of Def. 5.5.13, then so must a subtree of it rooted at a node in $\mathrm{Res}(\text{co-path}(\mathrm{ID}))$.

Moreover, by assumption, we know that $\mathrm{PHash.Ver}(T_q) = 1$. Recall this means that every non-blank node in the T_q has a complete public state; and that for any internal node w, its associated ID_v is that of a user whose leaf is in the sub-tree rooted at w, and if w_1 and w_2 are w's effective parents, we have that either:

(a) $h^2_{w_1} = \mathsf{H}_4(\mathrm{pk}_w, \mathrm{PK}^{\mathrm{pr}}_w, h^2_w, \{\mathrm{pk}_y\}_{y \in R})$ and $h^1_{w_1} = \ell(w_2)$ or

(b) $h^2_{w_1} = \mathsf{H}_4(\mathrm{pk}_w, \mathrm{PK}^{\mathrm{pr}}_w, h^2_w, \mathrm{PK}^{\mathrm{pr}}_{w_2})$ and $\mathcal{H}_{trans,w_1} = \mathcal{H}_{trans,w_2}$.

for $R = \mathrm{Res}(w_2) \setminus \mathrm{Unmerged}(w)$.
Moreover, $\mathrm{SIG.Ver}_{\mathrm{svk}_w}((\mathrm{pk}_w, \mathrm{PK}^{\mathrm{pr}}_w, h_w, \mathcal{H}_{trans,w}, \mathrm{confTag}_w), \sigma_w) = 1$.

Let ID_1 be the user located at the end of the path from a ratchet tree leaf in T_q to v which, again, we know exists by assumption. For ease of exposition, we will assume

w.l.o.g. that ID_1 is the left-most leaf in the subtree of $\mathcal{T}^q$ under v .(where $\mathcal{T}^q$ is the ratchet tree of the whole group in the view of ID; even though ID does not have a full view of $\mathcal{T}_q$, they do know which users are associated to which leaves.) ID_1's path in T_q will contain a number of updates from some of the users under v's subtree, which allow us to partition said path as follows. Let v_1 be the leaf of ID_1, and $v_1, \ldots, v_k = v$ the nodes in said path. As noted above, every key associated to the v_i must stem from either an add 9in the case of v_1 or an update query, since all the signature at each v's state verifies. We can partition this path into sub-paths given by sets of nodes that were sampled in the same update: v_i and v_{i+1} are in the same sub-path exactly when v_{i+1} verifies through v_i, according to $\mathrm{PHash.Ver}$. We denote the different updates that make up said path, and such that each corresponds to one element in the partition, as $q_1, \ldots, q_m$, in order, with q_1 being the update setting the key for v_1, and q_m the one setting the key for v. Accordingly, we denote by ID_i the user who authored q_i. Let q_i be an update corresponding to a partition element with two or more nodes, i.e. to which two or more nodes in the path correspond to. Then, we denote by $q_{i,j}$, $j \in (1, \ldots, m_i)$ the updates that correspond to the nodes that are in $\mathrm{Res}(\text{co-path}(\mathrm{ID}_1))$, i.e. in the resolution of ID_1's co-path, which for which their child corresponds to q_i. If there any blank nodes in between those sampled by q_i and q_{i+1}, we denote by $q^b_{i,j}$ those updates corresponding to nodes in the resolution of said blank nodes, and which are not q_1. Here $j < j'$ if $q_{i,j}$ (resp. $q^b_{i,j}$) was sampled by a party whose index is to the left of the party who sampled $q_{i,j'}$ (resp. $q^b_{i,j'}$). As before, we denote the user who sampled $q_{i,j}$ (resp. $q^b_{i,j}$) by $\mathrm{ID}_{i,j}$ (resp. $\mathrm{ID}^b_{i,j}$).

Along the way we will distinguish based on whether the parent hash of the lowest node in $\mathrm{path}(\mathrm{ID}_1)$ sampled by any given q_i verifies through conditions (a) or (b) above. Recall that condition (a) corresponds to the case where ID_i was aware of q_{i-1} at the time q_i, and had already weakly processed it by that time (as we will now show). In turn, condition (b) corresponds to the case where q_i and q_{i-1} were concurrent, in the sense that both users were in consistent states when sampling them and were not aware of the existence of the opposite update when sampling theirs. Thus, for ease of notation, we will write $q_{i-1} \triangleleft q_i$ to represent the first case, and $q_{i-1} \sim q_i$, to represent the second one. For any query $q_{i,j}$, i.e. those affecting only the nodes on v's copath, we say, similarly, that $q_{i,j} \triangleleft q_j$ and $q_{i,j} \sim q_j$ if the node $\mathrm{Int}(\mathrm{ID}_i, \mathrm{ID}_{i,j})$ verifies through condidions (a) and (b), respectively.

We will now prove the following claim, from which the Lemma will follow easily. Informally, it states that from its state at q_1 onwards, there is a chain of round messages that ID_1 would accept and that would bring them to a state consistent with ID's current state, along the way explicitly processing all updates corresponding to nodes in the path and copath of ID_1.

Claim. *For any $j \in (1, \ldots, m-1)$, such that $q_j \triangleleft q_{j+1}$, there is a sequence of efficiently*

computable round messages $M_1, \ldots, M_{r_j}$ that ID_1 would process and accept with respect to their state at q_1, and such that the state of ID_1 after processing that last message M_{r_j}, $\gamma_{\mathrm{ID}_1}^{q_{r_k}}$, is consistent with that of ID_{j+1} at time q_{j+1}, i.e. with $\gamma_{\mathrm{ID}_{j+1}}^{q_{j+1}}$.

Proof of Claim. First, we make the following observation: if $\tilde{q} \triangleleft \hat{q} < q$, then it must be that the user $\hat{\mathrm{ID}}$ that issued the update in $\hat{q}$ had already weakly processed $\tilde{q}$ at the time $\hat{q}$. Indeed, by the fact that parent hash verifies at the node where the both update paths meet, we know that $\hat{\mathrm{ID}}$ had some key from $\tilde{q}$ in their state. If there are not blank nodes in between the nodes set by $\hat{q}$ and the node set by $\tilde{q}$, then the update in $\tilde{q}$ must have been explicitly processed by $\hat{\mathrm{ID}}$, since a node they sampled belongs to the copath of $\hat{\mathrm{ID}}$. If, instead, there is at least one blank node in between $\hat{q}$ and $\tilde{q}$ and $\hat{(\mathrm{ID})}$ did not explicitly process $\tilde{(q)}$ at any point, then, by the minimality of q, it must be that $\hat{\mathrm{ID}}$ had already weakly processed $\tilde{q}$ at $\hat{q}$. To see why, note that $\hat{\mathrm{ID}}$ introduced the node corresponding to $\tilde{q}$ into their state at some point before $\hat{\mathrm{ID}}$, by processing a round message containing at least one remove operation. However, since $\hat{\mathrm{ID}}$ accepted that round message, it must be that the parent hash verification of said tree passed and, moreover, that it was a well-formed tree containing at least one user leave (that of the removed party). Further, all node states in it must have corresponded to past queries, except with negligible probability, since the adversary is not allowed use of signing keys and all node states come with a signature. Thus, that tree would have satisfied all constraints in Def. 5.5.13 and, moreover, have contained a node state corresponding to an operation not yet weakly processed by $\hat{\mathrm{ID}}$. In particular, this means A could have won the OPEN game with respect to some query earlier than q, which is a contradiction to the minimality of q.

We will prove the claim by induction on j, but will first start with a couple of simplified case, capturing most of the arguments used in the inductive argument, in order to build intuition. Suppose first $j = 1$, i.e. $q_1 \triangleleft q_2$, and we assume that there are no blank nodes between the nodes sampled by q_1 and those sampled by q_2. We will show the claim is true for this particular case.

We first show that ID_2 was at some point in a state consistent with that of ID_1 at q_1. To see this, first note that at least one node state set by q_1 is part of ID_2's state at q_2, from parent hash verification. Now, if q_1 corresponds to the query where ID_1 processed their welcome message, then the state of ID_1 after processing the corresponding round message must have been consistent with that of ID_A, the party who added them, at the time they issued the add operation. However, since ID_2 also must have processed the corresponding add query from ID_A, their state must have also been consistent with ID_A's at that time and, by transitivity of $=$, also with ID_1's at q_1. If, in turn, q_1 was sampled by ID_1 after they joined the group, then we know it must have been explicitly processed by ID_2 (since by assumption there are no blank nodes between nodes from q_1

and nodes from q_2, i.e. the highest node sampled by q_1 is the parent of a node sampled in q_2 and therefore a node in the copath of ID_2), which, by Proposition 3, means their state before processing it was consistent with that of ID_1 at q_1.

Now, from the time the state of ID_2 was consistent with ID_1's at q_1 all the way to q_2, ID_2 must have processed a series of round messages (we know they processed at least one: the one containing q_1). For any $q_{1,j}$ such that $q_{1,j} \lhd q_1$, we know that ID had already weakly processed $q_{1,j}$ at q_1, by the observation at the beginning of the claim's proof. Now, note that all packets from the $q_{1,j}$ such that $q_{1,j} \sim q_1$ will be accepted by ID_1 with respect to its state at q_1, since by parent hash verification of nodes sampled by q_1 (in particular the check involving $\mathcal{H}_{trans}$), ID_1 was in consistent states at that time with the corresponding user $\mathrm{ID}_{1,j}$, and moreover, by the equality check involving the predecessor keys $\mathrm{PK}^{\mathrm{pr}}$, ID considered those parties as belonging to the corresponding subtree. This is necessary, as ID_1 will only accept an packet corresponding to some node u if it is signed by a user belonging to the subtree under u. Note that if $\mathrm{PK}^{\mathrm{pr}}$ was not part of the state or not included in the signatures, a tree resulting from rearranging update paths from users that update concurrently and from consistent states, would pass parent hash verification. For example, consider the case where all users update concurrently, with the left update always winning; the tree where the update paths of any even-index users are rearranged would still verify (this issue is similar as the one that earlier versions of parent hash for TreeKEM suffered of [AJM22]).

Consider now all other packets included in the round message $M_{\mathrm{ID}_2,1}$ that ID_2 processed and through which they processed q_1, which were either adds or removes, or corresponded to node states for nodes that are either $\mathrm{Int}(\mathrm{ID}_1, \mathrm{ID}_2)$, nodes above it, or parent of these. All this packets would also be accepted by ID_1 with respect to $\gamma_{id_1}^{q_1}$, since they were accepted by ID_2 at a time where their state was consistent with it; and the information used to determine aceptance of this packets is based on values that lie in the intersection of the states of both parties. Indeed, the acceptance of update or add packets is based uniquely on values corresponding to the key schedule and membership set, and the acceptance of removes is based also on values stored on node states. Since the nodes mentioned are in the states of both parties, and the node states must match by the fact that the states are consistent, the statement follows. Thus, the message M_1 containing the packets corresponding to the $q_{1,j}$ such that $q_{1,j} \sim q_1$, and all other packets from $M_{\mathrm{ID}_2,1}$ that are either adds, removes or correspond to the node states above will be accepted by ID_1. Moreover, if this message contains, for packets corresponding to the $q_{1,j}$, the same openings as the corresponding nodes in T_q, and, for all other node states, the same openings included in the message $M_{\mathrm{ID}_2,1}$ to ID_2, then ID_1 after processing M_1 will be in a state consistent with ID_2 after they weakly processed q_1. To see this, note that the states of nodes in ID_1's state below $\mathrm{Int}(\mathrm{ID}_1, \mathrm{ID}_2)$ will match exactly those in T_q, and that those for the remaining nodes must match those in ID_2's state. Since by parent hash verification, we know $h^1_{w_2} = \ell(w_1)$, where w_1 and w_2 are the

left and right parents of $\mathrm{Int}(\mathrm{ID}_1, \mathrm{ID}_2)$, we know the label both parties compute for w_1 must be the same as well.

As to the remaining round messages processed by ID_2 from this time until q_2, it must be that none of this included any changes to the subtree under the highest node sampled by q_1. Indeed, since we are for now assuming that there are no blank nodes in between those sampled by q_1 and q_2, for ID_2 to process something affecting that node after processing q_1, they would have to explicitly process q_1 again in order to have a node state from that query in their state at q_2. However, by Corollary 5.5.9.1, we know this cannot be the case, since explicitly process implies weakly process. Thus, all other messages processed by ID_2 from that point onwards until q_2 must have only affected nodes that are, are above, or are parents of $\mathrm{Int}(\mathrm{ID}_1, \mathrm{ID}_2)$. By the same argument as before, ID_1 would process and accept the corresponding round messages $M_2, \ldots, M_{r_1}$, containing those same packets, in the same order, bringing them to a state consistent with ID_2's at q_2. This proves the claim above for simpler case where $j = 1$, there are no blank nodes between the nodes sampled by q_1 and q_2.

Before we move on, we will now discuss the effect of blank nodes between the nodes sampled by q_1 and q_2 in the above argument. First note that now we do not have the guarantee that ID_2 explicitly processed q_1, however this is taken care of by the observation from the beginning of the claim, which implies ID_2 weakly processed q_1 before q_2. By Proposition 3, this implies ID_2 was at some point in a state consistent with that of ID_1 when issuing q_1.

Further, we now need to show that ID_2 did not process any other operations affecting nodes on the subtree under the highest node sampled in T_q by q_1. Above we relied on the fact that ID_1 would have to explicitly process q_1 twice but, again, we do not have this guarantee. We argue as follows. In the case wjere blanks between q_1 and q_2, if ID_2 weakly processed but did not explicitly process q_1, say by processing round message $M_{\mathrm{ID}_2,1}$, it must be that they processed and accepted a round message containing some removal *after* they processed $M_{\mathrm{ID}_2,1}$. Let $M_{\mathrm{ID}_2,2}$ be some round message, processed and accepted by ID_2 between weakly processing q_1 (through processing $M_{\mathrm{ID}_2,1}$) and issuing q_2, where ID_2 weakly processed such an operation affecting the nodes on the subtree under the highest node sampled in T_q by q_1. First, note that if ID_2 processed no round messages including removals between processing $M_{\mathrm{ID}_2,2}$ and time q_2, then they must have processed some round message in that interval setting the key of some node in $\mathrm{Res}(\mathrm{lparent}(\mathrm{Int}(\mathrm{ID}_1, \mathrm{ID}_2)))$ to one set by q_1. But this node must have already been part of ID_2's state at that time, meaning they would have explicitly processed q_1 after already weakly processing it, a contradiction to Corollary 5.5.9.1. Thus, ID_2 must have processed some round message containing a remove after processing $M_{\mathrm{ID}_2,2}$ and, in particular, it must have been such a message that put the key from q_1 into their state. However, for ID_2 to accept such a message

the received tree corresponding to the removed user(s) path and co-path resolution must have been consistent with said key from q_1. Since we know the openings left by $M_{\mathrm{ID}_2,2}$ were not consistent with it, ID_2 must have processed yet another $M_{\mathrm{ID}_2,3}$ in between these two messages, in particular including openings now again consistent with the key from q_1. Assume $M_{\mathrm{ID}_2,3}$ is the last such round message processed by ID_2 before they the aforementioned round message that put the key from q_1 into their state. We know that that round message containing a remove must have contained a tree T that was consistent with the openings sent in $M_{\mathrm{ID}_2,3}$ (and thus such that all queries corresponding to node states in T had already been weakly processed by ID_2 at the time, by the argument at the beginning of the claim's proof). Now T must contain some query q' that ID_2 weakly processed between (and including) $M_{\mathrm{ID}_2,2}$ and $M_{\mathrm{ID}_2,3}$, such that $q_1 \triangleleft q'$ in T. If there are no blank nodes between q' and q_1, then whoever issued q' must have explicitly processed q_1, implying ID_2 weakly processed q_1 a second time in that interval, a contradiction. Else, we can recurse and repeat the same argument with respect to ID', which must eventually lead us to some user weakly processing q_1 twice, since we are considering progressively lower nodes in the tree.

The remaining part of the argument that is affected by these blank nodes is that now there are queries $q^b_{1,j}$ that we need to show ID_1 would weakly process by the time they process M_{r_1}. To see this is the case, note that all such blank nodes between q_1 and q_2 must have been created (in the view of ID_2) between (possibly including) them processing q_1 and the query q_2. Moreover, during this interval, ID_2 cannot have processed any removes coming from the subtree under the highest node sampled by q_1 in T_q, following the argument above. However, note that if we were to construct the sequence of round messages for ID_1 as before, we would now need to include any updates affecting nodes in the subtree under $\mathrm{Int}(\mathrm{ID}_1, \mathrm{ID}_2)$ but outside the subtree under the highest node sampled by q_1. Further, note that these might no longer be reflected in T_q, since there could have been several of them, and earlier ones could have been overwritten. Nevertheless, we can use the algorithm E from Lemma 5.5.15 to determine which operations affecting these nodes were processed by ID_2 in this time interval, and add those to the crafted messages to ID_1, with the corresponding openings. Further, tree that the server would need to send to ID_1 for each message containing a remove would contain exactly the same node states as the tree received by ID_2 in their corresponding messages. Thus, they would be accepted by ID_1. In particular, any of the nodes corresponding to the $q^b_{1,i}$ either corresponds to an update ID_2 explicitly processed (either after or in one of the messages where the blanks were created), or would have been included in at least one of such trees. In the first case, ID_1 would also explicitly process such query in the process; in the second, again, by minimality of q, we know that ID_1 must have weakly processed the corresponding query. This proves the claim for $j = 1$ in the general case, where blank nodes can exist between q_1 and q_2.

Now, we will start the induction proof of the claim. Let $j \in (2, \ldots, m-1)$ and suppose that $q_j \triangleleft q_{j+1}$ and, further, that for any $j' < j$ for which $q_{j'} \triangleleft q_{j'+1}$, a sequence $M_1, \ldots, M_{r'_j}$ as in the claim exists. For the base case, we consider the case where j is the minimal such index for which $q_j \triangleleft q_{j+1}$. In that case all, for all $j' < j$, $q_{j'} \sim q_{j'+1}$, and thus, by parent hash verification (in particular by the fact that the transcript hash values in those updates match), we know that all $\mathrm{ID}_{j'}$ were consistent with each other with respect to their respective states $\gamma^{q_{j'}}_{\mathrm{ID}'_j}$. In particular, it means that ID_1 would accept any packet from those updates that was included in a round message. Moreover, as above, we will show that ID_{j+1} must have weakly processed q_1 at some point, which in turn implies that ID_{j+1} must have at some point had a state consistent with that of ID_1 at q_1. To show this, note that by parent hash or verification of the node $\mathrm{Int}(\mathrm{ID}_{j+1}, \mathrm{ID}_1)$, we know that, at q_{j+1}, the label of the left parent of said node, in the view of ID_{j+1} corresponded exactly to the label of that subtree of T_q, to which the node(s) sampled by q_1 belong to. By minimality of q, we know then that ID_{j+1} must have weakly processed q_1.

Similar to above, M_1 can now be constructed as containing all $q_{j'}$ for $j' \in \{1, \ldots, j\}$, together with all the $q_{j',\hat{j}'}$ such that $q_{j',\hat{j}'} \sim q_{j'}$, as well as all queries contained in the message that ID_{j+1} processed when they weakly processed q_1, and which affected nodes outside of the subtree under the highest node sampled by q_j. A similar argument to the one above shows that ID_1 would accept all these individual packets and thus the round message, and, if this round message contained the correct openings (those given by T for nodes the subtree under the highest node sampled by q_j, and those given by the round message for ID_{j+1} for the remaining nodes), then ID_1 will transition to a state consistent with that of ID_{j+1} at the time the latter processed q_1. The exact same argument above shows that there is a sequence $M_2, \ldots, M_{r_j}$ of round messages ID_1 would process and accept after processing M_1 that bring them to a state consistent with that of ID_{j+1} at q_{j+1}. Moreover, along the way, ID_1 processes all $q^b_{j,\hat{j}}$ that they had not already weakly processed, as before, those would correspond to nodes whose states must have been included by the server in the corresponding round messages including the removals that caused any blank nodes between the nodes sampled by q_j and those sampled by q_{j+1} (note that no such $q^b_{j,\hat{j}}$) exist if no such blank nodes exist). Finally, just as before, ID_1 must also before q_1 have processed any $q_{j',\hat{j}}$ for $j' \leq j$ for which $q_{j',\hat{j}} \triangleleft q_{j'}$. Indeed, we know the label of the highest node sampled by q_j in T matches the label of the same node in ID_1's state after processing q_1, so by minimality of q, ID_1 must have processed all queries corresponding to this subtree. This concludes this case.

Now, we consider the inductive step, where j is not minimal, i.e. there exist queries q_κ satisfying $q_\kappa \triangleleft q_{\kappa+1}$ with $\kappa < j$, and we assume that the claim is true for any such $\kappa < j$. Let $q_{j'}$ be the query where $j' < j$, $q_{j'-1} \triangleleft q_{j'}$, and such that j' is maximal satisfying

those two properties. In particular, j' is such that for all $\hat{j} \in \{j'+1, \ldots, j-1\}$ (if such interval exists) we have $q_{j'} \sim q_{\hat{j}}$. By the induction hypothesis there is a sequence of messages $M_1, \ldots, M_{r_{j'-1}}$ which ID_1 would process with respect to their state at q_1 and which ushers them into a state consistent with that of $\mathrm{ID}_{j'}$ at time $q_{j'}$.

In this case, using the same argument as above, ID_{j+1} processed all $q_{j'}, \ldots, q_j$, by minimality of q, before q_{j+1}. Thus, ID_{j+1} was at some point in the past in a state consistent with $\mathrm{ID}_{j'}$ at $q_{j'}$, and thus with ID_1 after they processed $M_{r_{j'-1}}$. Moreover, it must be that since that point and all the way to q_{j+1}, ID_{j+1} has not weakly processed any updates, adds or removes affecting nodes in the subtree under the left parent of $\mathrm{Int}(\mathrm{ID}_1, \mathrm{ID}_{j'})$. To see why, note first that all the $q_{j'}, \ldots, q_j$, must have processed at the same time since they all had the same transcript hash values. At the time just before processing those, $q_{j'}$ had to have already weakly processed all updates below $q_{j'}$. Indeed, we know that there is a tree that would agree with $\mathrm{ID}_{j'}$'s state at that time and which would contain the nodes for all those queries (following the fact that the parent hash verification of $\mathrm{Int}(\mathrm{ID}_1, \mathrm{ID}_{j'})$ is through condition (a)). But since the state $\mathrm{ID}_{j'}$ had at time $q_{j'}$, is consistent with ID_j's state at the time they processed $q_{j'}$, such tree must have also been consistent with ID_{j+1}'s state, showing, again, by minimality of q, that ID_{j+1} had weakly processed those queries by then. Moreover, we know that at q_{j+1} their state was also consistent with the subtree under the highest node affected by q_j in T_q. Thus, if ID_{j+1} processed anything else in between weakly processing $q_{j'}$ and time q_{j+1} affecting those nodes on the subtree under the highest node affected by $q_{j'}$, then must have processed some other round message afterwards that brought their state consistent with the subtree under q_{j+1} again. In particular, they must have processed something that affected the left parent of $\mathrm{Int}(\mathrm{ID}_1, \mathrm{ID}_{j+1})$, and later processed something that set the key at the highest node sampled by q_j in T to be the same one as they already had in their state at the time of weakly processing $q_{j'}$ (and therefore also q_j). This means that they must have weakly processed q_j twice, which is a contradiction to Corollary 5.5.9.1. Thus, it must be that since the time ID_{j+1} weakly processed $q_{j'}$ and all the way to q_{j+1}, ID_{j+1} has not weakly processed any updates, adds or removes affecting nodes in the subtree under the left parent of $\mathrm{Int}(\mathrm{ID}_1, \mathrm{ID}_{j'})$.

In particular, this means that we can treat this as the case where j is minimal, and argue that there is a sequence of round messages $M'_1, \ldots, M'_{r'_j}$ that $\mathrm{ID}_{j'}$ would process from time $q_{j'}$ that would put them into a state consistent with ID_{j+1} at q_{j+1}, where $\mathrm{ID}_{j'}$ would along the way weakly process all queries corresponding to nodes between $q_{j'}$ and q_j; and, moreover, such that those messages would contain no changes from the subtree under $q_{j'}$. But then, the corresponding sequence of round messages $M^1_1, \ldots, M^1_{r'_j}$, sent this time to ID_1 after they processed $M_{r_{j'-1}}$, and containing the same packets as the $M'_1, \ldots, M'_{r'_j}$, would have the same effect and would also be

accepted, since they only concern nodes that in the intersection of the states of the two parties. Thus, if we let $M_{r_{j'-1}+k} \leftarrow M_k^1$ for $k \in \{1, \ldots, r'_j\}$ then the sequence of messages $M_1, \ldots, M_{r_{j'-1}+r'_j} =: M_{r_j}$ brings ID_1 from their state at q_1 to a state consistent to ID_{j+1}'s at q_{j+1}, along the way weakly processing all the necessary queries. This completes the proof of the claim.

□*(End of proof of Claim)*

If q_j is the maximal query in ID_1's path in T_q for which $q_{j-1} \triangleleft q_j$, then we know from the proved claim above that there is a sequence of round messages that ID_1 would process from its state in q_1 that lead them to a state consistent with that of ID_j at q_j. Moreover, for any $j' > j$, $q_{j'-1} \sim q_{j'}$, just by definition of q_j. Thus, we will argue now that there exists one last message M_m that ID_1 would process with respect to their state after processing $M_{r_{j-1}}$, and which will take them to a state consistent with that of ID at the time they weakly processed q_m. First, observe that ID's state before weakly processing q_m, in particular the corresponding openings, must have been consistent with the subtree of T_q below q_j, since they must have been in consistent states with ID_m at q_m and therefore with ID_j at q_j (since $q_j \sim q_m$). Moreover, after processing the round message through which they weakly processed q_m, their state must still be consistent with said subtree of T_q, just because of the assumption that $\ell(T)$ is equal to the label $\ell(v)$ is ID's state. This implies that this round message through which they processed q_m contained no changes for that subtree of T_q in question, following the same argument used above. Thus, following the arguments in the proof of the claim, the round message, now sent to ID_1 containing the same packets as the one sent to ID would be accepted and would prompt ID_1 into a state consistent with that of ID.

However, this implies that all queries $\tilde{q}$ in T_q such that $\tilde{q} \sim q_j$ would be explicitly processed by ID_1 by processing this message, and thus weakly processed by ID at this time. Moreover, all other queries corresponding to all other nodes in T_q were already weakly processed by ID_1 at the time before they processed M_m. In particular, since the states of ID and ID_1 before they each weakly processed q_j were consistent, so must have been their round hash value at that round. Thus, by minimality of q, again, ID must have already weakly processed all those queries by that time. This concludes the proof. □

Finally, we turn to the proofs of Lemmas 5.5.10 and 5.5.11.

Proof of Lemma 5.5.10. Before starting the proof we make the following observation. Let ID and ID^* be in consistent states after each processing queries $q = \mathrm{CGKA.Proc}(\gamma_{\mathrm{ID}}, M)$ and $q^* = \mathrm{CGKA.Proc}(\gamma_{\mathrm{ID}^*}, M)$, respectively, and assume that ID^* joined the group at the same time or after than ID. Then, by collision resistance of H_5, the sequence of transcript hash values (and round hash values) ID^* has computed

throughout their execution, up to time q^*, must be a suffix of those ID has computed up to time q. In particular, for any query of the form $\mathrm{CGKA.Proc}(\gamma^*_{\mathrm{ID}}, \cdot)$ in the protocol execution where ID^* accepts the received round message, there must be a query $\mathrm{CGKA.Proc}(\gamma_{\mathrm{ID}}, \cdot)$ where ID accepts the received round message; and moreover, their states between any two pairs of such corresponding queries must be consistent, by Proposition 2. Indeed, this must be the case since accepting and processing a round message always changes the users' state, as e.g. the transcript hash is updated by hashing in a new round hash value. We can thus draw a 1-to-1 correspondence between processed messages by one and the other. More formally, if $(q^*_1, \ldots, q^*_k)$ are the process queries $q^*_i = \mathrm{CGKA.Proc}(\gamma^{q^*_{i-1}}_{\mathrm{ID}^*}, \cdot)$ that ID^* processed since joining the group up until q, then there is a bijection $q^*_i \mapsto q_i$ between those and the k last process queries $(q_1, \ldots, q_k)$, $q_i = \mathrm{CGKA.Proc}(\gamma^{q_{i-1}}_{\mathrm{ID}}, \cdot)$, that ID processed before q, such that the states of ID and ID^* between q_i and q_{i+1} and between q^*_i and q^*_{i+1}, respectively, are consistent.

We will start by arguing that every update U that ID^* weakly processed at or before time q was also weakly processed at or before time q by ID. Consider first the case where ID^* weakly processed U at the moment of joining the group, i.e. ID the tree T that ID received as part of the welcome message contained a node with a public key which was part of U. Because ID^* accepted the welcome message, we know that $\mathrm{PHash.Ver}(T) = 1$, and that it has the format required in Def. 5.5.13, that is, indegree two everywhere except at the leaves, no blank leaves, the state at each node corresponds to some update sampled during the protocol execution (since the signatures at each node verify), and at least one of its leaves contains a public signing key. Moreover, since the state of ID^* after processing q^*_1 is consistent with that of ID after processing q_1, it holds that the label of T must match the label of v_{root} in ID's state, $\gamma^{q_1}_{\mathrm{ID}}$, at that time. However, if T contained a node state corresponding to some query not yet processed by ID, we could then build an adversary against the OPEN game that would win using T with respect to query where ID processed q_1. It follows that ID had at that time already processed U.

Next, consider the alternative case, where ID^* weakly processed an update U after joining the group, through query q^*_i, $i \geq 2$. By definition, there exists a user ID' and a series of efficiently computable round messages $M'_1, \ldots, M'_t$ such that ID', with respect to their state γ' at time q'_{ID^*}, would process (and accept) all M'_i in order; and such that ID' would explicitly process U by processing M'_t and arrive at a state consistent with $\gamma^{q^*_i}_{\mathrm{ID}^*}$. Here, recall q'_{ID^*} is either $q'_{\mathrm{ID}^*} = \mathrm{CGKA.Upd}(\gamma')$ the last update from ID' weakly processed by ID^* or, if such update does not exist, the query in which ID' processed their welcome message into the group, initializing state γ'. Now, note that since the state of ID at q_i is consistent with that of ID^* at q^*_i, then so it is with the state of ID' after processing M'_t. We will show that there must exists a similar

sequence of round messages from q'_{ID}, the query where ID' last issued an update weakly processed by ID or, if such update does not exist, the query in which ID' processed their welcome message into the group. Assume first that q'_{ID^*} is the query where ID' processed their welcome message. In that case, $q'_{\mathrm{ID}^*} \leq q'_{\mathrm{ID}}$, so (a subsequence of) the M'_i satisfy the definition with respect to ID, meaning ID weakly processed U in q_i. If that is not the case, ID^* must have weakly processed query q'_{ID^*} at or after joining. If ID^* did so at the moment of joining, then we know that ID must have also weakly processed said query, so $q'_{\mathrm{ID}^*} \leq q'_{\mathrm{ID}}$ (in fact, equality holds here) so, again, the M'_i satisfy the definition with respect to ID. Last, assume ID^* weakly processed q'_{ID^*} in q^*_i, with $i \geq 2$. But then, we know that the state of ID' at q'_{ID^*} is consistent with ID^*'s state, $\gamma^{q^*_{i-1}}_{\mathrm{ID}^*}$, and therefore also with ID's, $\gamma^{q_{i-1}}_{\mathrm{ID}}$. Thus the M'_i, preceded with whatever round messages ID' processed between q'_{ID} and q'_{ID^*} satisfy the definition with respect to ID^*.

Finally, this last argument shows, in fact, that if ID^* weakly processes U in q^*_i, for $i \geq 2$, then so does ID in q_i. This completes the last part of the lemma statement, regarding the ordering of weakly processed queries (and, in particular, implies that $q'_{\mathrm{ID}^*} = q'_{\mathrm{ID}}$ in the last case above). □

Proof of Lemma 5.5.11. There are three actions that trigger ID to add a key to their local state: processing an add operation, and adding the new InitKey of the added party; explicitly processing an update operation; and processing a remove operation, where they add extra keys to the resolution of their copath. Statement 1 is trivially true for keys added through the first or second action. We will first show that statement 2 is also true for these keys.

Consider now the case where ID added pk_v to their state by processing an add operation, adding party ID_A, and assume for contradiction that they weakly processed an update U' affecting v in query $q' \in \{q+1, \ldots, q^*\}$. This update must be authored by ID_A, but since their leaf v is in the state of ID, they must have then explicitly processed it, which would mean v would no longer be in ID's state, a contradiction.

Now, consider the case where pk_v was introduced in ID's state by explicitly processing update U_v in q and, again, assume for contradiction they weakly processed some U' in $q' \in [q+1, \ldots, q^*]$ affecting v. For pk_v to still be part of ID's state at q^*, they must have not explicitly processed any update coming from the subtree under v. However, by the definition of weakly processing, there must be a user $\hat{\mathrm{ID}}$ under v's subtree whose update was explicitly processed by ID in q', which is a contradiction.

Thus, it remains to show the lemma for keys pk_v that were added by ID to their state by processing a remove operation. This is, however, a straight forward application of Lemma 5.5.14. Observe that pk_v must have then been part of a tree sent by the server which satisfied all conditions in Def. 5.5.13, because ID accepted them round

message containing it. Indeed, the checks a party does on this tree before accepting the round message that contains it ensure that it satisfies the requirements estipulated in the lemma: parent hash verifies and, by virtue of consisting of the path(s) and co-path resolution(s) of one (or more) user(s), it will have no blank leaves, indegree two except at the leaves, and contain a leaf with a public verification key. The fact that all node states correspond to previous queries follows, in turn, by the fact the adversary is not allowed to corrupt signatures. Moreover, its label must match that of the intersection of the removed user(s)'s path(s) and $\mathcal{P}(\mathrm{ID})$. Thus, it must be that all keys in it correspond to operations ID had already weakly processed. If that was not the case, we could build an adversary for the OPEN game which would win using the mentioned tree. with respect to ID and the tree(s) received from the server as part of the round message.

It remains to show that statement 2 is true for v, i.e. that ID cannot have weakly processed an update U^+ coming from a user in the subtree under v, after weakly processing U. However, note that if this is the case, ID must have explicitly processed an update between q and q^*, and later process another round message again making them weakly process U twice, following the same argument as in the proof of Lemma 5.5.14, which is a contradiction to Corollary 5.5.9.1. □

CHAPTER 6

DeCAF

6.1 Introduction

As we discussed in the previous Chapter,[1] while updates in the initial versions of TreeKEM only need $\log(n)$ communication, they are inherently sequential: a user can only send an update request after processing the previous one. If two (or more) users A and B send an update request each re-keying their full paths to the server for the same previous ratchet tree state (as shown on the left in Figure 6.1), the server will simply reject all but one of the requests. In fact, this is true for all CGKA variants with two exceptions discussed below.

Recall that recent versions of TreeKEM do allow for a different type of concurrent updates through the "Propose and Commit" framework. Concurrent updates, however, will not re-key the paths of both users, and will instead blank the paths of users proposing. Figure 6.1 (right side) shows the tree we get if A commits to an update proposal by B in this way. Note that the more concurrent updates, the more blanking occurs, and the more expensive future operations become e.g., to commit A must send 4 ciphertexts before blanking B, but 6 after. In general the cost can grow from $\log(n)$ to n. If the group members want communication efficiency, they will have to commit to as few updates as possible at a time, relying instead on *sequential* commits to refresh keys. That means concurrency is not possible anymore, as commits need to be totally ordered, and the issue outlined above returns.

[1] This Chapter essentially replicates, with permission, large parts of the full version [AAN+22a] of our publication to appear at the 14th International Conference on Security and Cryptography for Networks (SCN 2024).

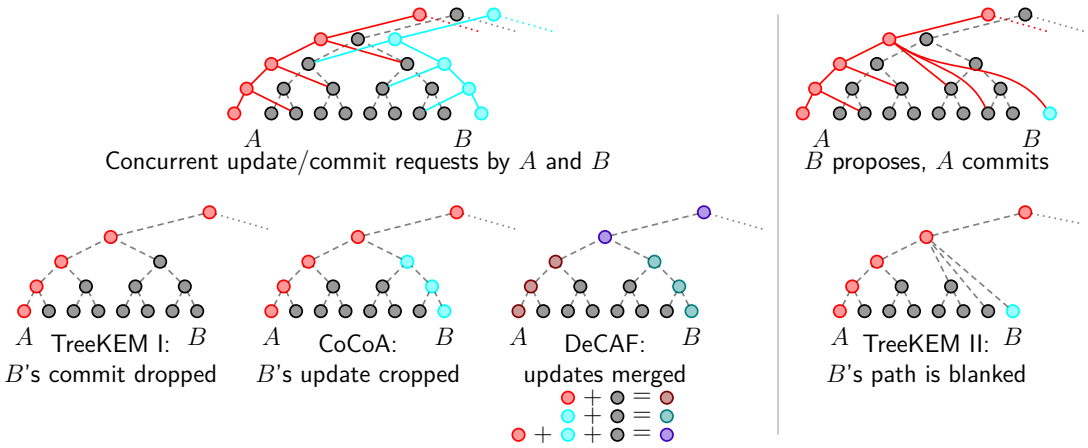

Figure 6.1: (left): Illustration of how TreeKEM, CoCoA, and DeCAF handle a concurrent update by parties A and B who want to replace their (potentially compromised) keys. TreeKEM I refers to the conservative approach where users commit one at a time. In DeCAF instead of replacing old keys, the new key-material is merged with the existing one. (right): An illustration of blanking used to commit an update proposal (removing B would be similar, with their leaf node blanked instead.)

Causal TreeKEM. The first CGKA protocol supporting concurrent updates was Causal TreeKEM [Mat19]. This protocol builds on a public key encryption primitive allowing for keys to be combined in a commutative way. This way, updates will no longer overwrite the previous key, but instead update it by combining the fresh key with the existing one. Since this combining process is commutative, several updates can me merged at the same time, without regard for the order in which users received them. As a downside, the protocol does not guarantee FS, and PCS requires a number of updates equal to the amount of corrupted group members, each of which needs to take place in a different round. Additionally, it does not formalize the security of the "key merging" functionality, and does not give full security proofs.

CoCoA. The protocol CoCoA [AAN+22b], which we presented in Chapter 5, processes concurrent update proposals in a "greedy" manner and simply accepts as many keys from concurrent updates as possible. As illustrated in Figure 6.1, if there is a conflict, i.e. two concurrent updates want to replace the same node, then one of the two updates is rejected *from this point upwards*. While this process does not guarantee that the key is safe after every compromised party updated,[2] somewhat surprisingly [AAN+22b] proves that the tree does heal after every party updated $\log(n)$ times in the worst case (independent of the number of compromised parties and allowing the adversary to schedule all operations and also decide which of any two concurrent updates "wins" in every case).

[2] In the example from Figure 6.1, if B was compromised, after the update the two topmost red nodes would still be compromised, as their keys were encrypted to compromised keys.

Moreover, CoCoA enjoys extremely low communication complexity, as each party must only download at most $\log(n)$ ciphertexts to process each set of concurrent updates. Note that this is independent of both the number m of parties that update in this round, which can be as large as $m = n$, as well as the number t of corruptions, which can be as small as $t = 1$. For this to be theoretically possible, the untrusted server must be more sophisticated than just relaying every protocol message it gets to all users in the group. Instead, it only sends a subset of the ciphertexts to each user based on their position in the tree, together with some commitment to its actions, allowing users to check that they received consistent messages.

Server- and blockchain-aided CGKA. In order to distribute protocol messages among the members of the group, CGKA protocols typically rely on an untrusted server. Most CGKA protocols like TreeKEM [BBR+23], rTreeKEM [ACDT20], and Tainted TreeKEM [KPPW+21] require a simple relay server. CoCoA, however, is a server-aided CGKA protocol, a primitive formally defined in [AHKM22], and where the server is expected to do non-trivial computation and provide users with personalized packages. This reliance on the server can still be problematic. For example, it allows it to reject protocol messages by a particular user, thus preventing them from healing. Or to selectively forward messages to only part of the users, leading to a group split.

These issues could be amended by replacing the server with a decentralized solution, an example of which would be a blockchain. Throughout the paper we will use the term blockchain for convenience to refer to any append-only data structure with the following property. When the data is distributed among multiple nodes there is a consensus mechanism that guarantees that the data is arranged into totally-ordered blocks that all nodes agree on. New data can be added by making use of a peer-to-peer network or any other suitable type of channels. The use of such an append-only structure (permissioned or permissionless) allows us to realize group messaging which enjoys the same robustness and security guarantees as the underlying structure. More concretely, instead of sending their CGKA protocol messages (update/add/remove) to the server, the users would post them on the append-only ledger. Only the key-management must be on-chain, text messages (encrypted under the current group key) can be gossiped or shared on a public bulletin board.

Note that any CGKA in the classical setting can be "compiled" to the blockchain setting: in the latter, the block producer simply emulates the server to compile the protocol messages that would be broadcast in the classical setting, and adds this message to the block. In the case of server-aided CGKA, after downloading all protocol messages stored on chain, the users can simply locally emulate the computation that would be done by the smart server. This potentially increases the download communication-complexity

though, as the users no longer receive personalized packages.[3] The opposite holds as well, any server being able to emulate the outputs of the decentralized consensus protocol.

There are at least three separate properties which are achieved in the decentralized setting, but not in the "classical" server setting. Namely (1) security against splitting attacks, (2) censorship resistance, and (3) robustness. Regarding (1), an attack which is unavoidable in the classical setting is a splitting attack, where the (corrupted) server splits the users into two or more groups, and then only relays messages within those groups, forcing parties in different groups into different and inconsistent states. With such an attack one can, for example, enforce that only a particular subset of users sees some set of messages. If the protocol messages are on a blockchain, all parties will agree on the same view, and thus this attack is prevented. With regards to (2), another attack that is unavoidable in the single server setting is the censoring of a particular party. The server can ignore messages from a party, this way preventing them from ever updating. This is severe as, should this party be corrupted, the corrupted key can be indefinitely prevented from healing. In the blockchain setting, the "liveness property" of the blockchain, in combination with the fact that our protocol allows for concurrent updates (so there are no denial-of-service-type attacks where some parties prevent another one from updating by flooding the mempool) prevents this attack: if a user wants to update, their request will be added with high probability within a few blocks. Finally, concerning (3), in the single server setting the group can be shut down by taking out a single server. Better resilience can be achieved with several servers, but then one needs to solve the state machine replication problem. This is what our protocol does if using a permissioned blockchain. With a permissionless blockchain, resilience would be even stronger.

Let us mention that in order to avoid all three issues mentioned above we need to record all the protocol messages on chain, which is probably no problem in the permissioned setting, but could be expensive in a permissionless blockchain. Permissionless blockchains like Bitcoin or Ethereum have slow block arrival rates (and even slower confirmation times), there also is a non-trivial cost incurred by recording transactions on chain. A permissioned blockchain, on the other hand, just requires a fixed small number of servers and provides the required security as long as a majority of the servers behave honestly (e.g., 3 out of 5). The cost of running such a protocol is only a small constant factor larger than just having a single server, but greatly reduces the trust required. If we are only interested in (1) and (2), but not (3), one can just post a single hash of all the messages which each block contains on chain, while the actual messages are

[3]While the use of smart contracts might seem like a natural strategy to address this, the suitability of this approach is not clear: who executes the smart contract? can one justify or avoid storing the server state on the blockchain?

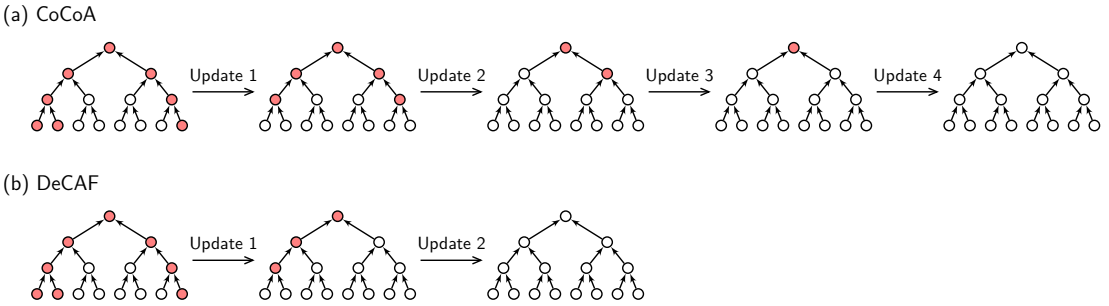

Figure 6.2: Comparison of the number of rounds required to recover in CoCoA (a) and DeCAF (b) for n users, of which t are corrupted. Red nodes correspond to compromised keys. In each round all parties update concurrently, in CoCoA update requests are prioritized from left to right. CoCoA requires $\lceil \log(n) \rceil + 1 = 4$ rounds to recover, DeCAF only $\lfloor \log(t) \rfloor + 1 = 2$.

stored off chain. This loses property (3) unless we solve the data availability problem separately[4].

6.1.1 Our Contribution

DeCAF. In this work we consider a new CGKA protocol, DeCAF (for DEcentralizable Continuous group key Agreement with Fast healing), that allows for concurrent updates. In DeCAF we use a key-updatable PKE scheme, and updates no longer *replace* keys, but *update* them. We show that the protocol provides forward security in the same vein as most other CGKAs (albeit slightly weaker than TreeKEM due to a potential delay until update messages are received and processed by other users), and only needs $\log(t)$ epochs to heal, with t being the number of corrupted parties. The latter point contrasts to CoCoA, where it is only guaranteed that the tree healed once each compromised party updated $\log(n)$ times. This difference is illustrated in Figure 6.2. The root of this difference is the fact that, while in CoCoA we must drop one of two concurrent updates for the same node, in DeCAF we can perform both, which turns out to have a significant impact on security. As we can expect t to be small compared to n (in fact, for most of the lifetime one should hope that $t = 0$), DeCAF will provide comparable security to CoCoA with fewer updates. On the downside, as in DeCAF every user must process all updates by other users (while in CoCoA at most $\log(n)$ other updates matter), the download communication (from server to users) will be larger.

The above discussion suggests a trade-off between DeCAF and CoCoA, and which one is better will depend on the context. If run using a server, CoCoA and DeCAF are incomparable; DeCAF heals faster ($\log(t)$ vs $\log(n)$ rounds) and therefore has lower sender communication, but CoCoA has lower recipient communication (since the server crafts individual messages for each party). However, in the decentralized setting (where

[4] https://blog.polygon.technology/the-data-availability-problem-6b74b619ffcc/

we do not want to rely on a(n intelligent) server to relay protocol messages), CoCoA loses its advantage in recipient communication and DeCAF is strictly better in all aspects as we discuss in more detail below.

Our protocol is also similar to Causal TreeKEM [Mat19] in some aspects, but differs largely in others. In particular, the main element in common is the above-mentioned use of updatable PKE, which is exclusive to these two protocols. While the primitive is also part of other constructions, such as rTreeKEM [ACDT20], it is employed in a very different way, as the focus is another (improved FS, in that case). However, while Causal TreeKEM requires the key-update functionality to be commutative, we do not. Furthermore, mechanisms for adding and removing parties are different, with those used by DeCAF being both simpler and in line with what is currently used by MLS, making a potential adoption by the standard much easier. Another big difference is the security guarantees provided by both protocols. Indeed, Causal TreeKEM does not consider FS, and PCS is only claimed after each corrupted user issues an update in a separate round, thus needing t rounds to heal. The latter claim lacks a formal security proof. We conjecture that for static groups, Causal TreeKEM might enjoy a similar PCS guarantee, but this is unclear for dynamic groups.

Maintaining a Group on Chain. Given the particular suitability of DeCAF in a decentralized network, we cast it as making use of a blockchain, access to which is shared by all group members. The use of blockchain for CGKA protocols is novel as far as we know, but note that there exist previous messaging protocols making use of it, like Elixxir [Coi]. We stress that this is not a requirement for the protocol to run, which could instead simply rely on a central server, as discussed above. Now we explain how to make use of such an structure to maintain a group. In its simplest instantiation, a group would be initialized once some ith block B_i in the blockchain contains the welcome messages which defines a ratchet tree T_i for some group. Users in the group can post add/remove/update messages on the blockchain, and the ratchet tree T_j is defined to be the ratchet tree T_{j-1} after processing the protocol messages contained in block B_j. One issue with this basic protocol is the fact that a message created referring to T_i can only be created after learning block B_i and must be added to the next block B_{i+1}. Depending on the block-arrival time of the chain, we might want to give messages more time to get included in the blockchain. We use a simple way to achieve this by introducing a parameter k, and only update the ratchet tree every k blocks, so messages referring to this tree can be included in any of the k blocks following the block specifying the tree. The parameter k should not be chosen larger than necessary, as only one update per k-block epoch will contribute towards healing (except if a corruption occurs in between two updates from the same epoch). If a message is not included in time this just means it can no longer be included, so the user can simply create a new message referring to the new ratchet tree.

Protocol	Conc.	Rounds	Sender comm.	Recipient comm.	Cost after rec.
TreeKEM I [BBR18]	No	n	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(\log(n))$
TreeKEM II [BBR18]	Yes	2	$\mathcal{O}(n)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
Causal TreeKEM [Mat19]	Yes	n	$\mathcal{O}(n\log(n))$	$\mathcal{O}(n\log(n))$	$\mathcal{O}(\log(n))$
Bienstock *et al.* [BDR20]	Yes	2	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$	$\mathcal{O}(\log(n))^*$
Weidner *et al.* [WKHB21a]	Yes	2	$\mathcal{O}(n^2)$	$\mathcal{O}(n)$	$\mathcal{O}(n)$
CoCoA [AAN+22b]	Yes	$\lceil\log(n)\rceil+1$	$\mathcal{O}(n\log^2(n))$	$\mathcal{O}(\log^2(n))$	$\mathcal{O}(\log(n))$
DeCAF (this work)	Yes	$\lfloor\log(t)\rfloor+1$	$\mathcal{O}(n\log(n)\log(t))$	$\mathcal{O}(n\log(n)\log(t))$	$\mathcal{O}(\log(n))$

Table 6.1: Overview of the cost incurred to heal t corruptions in a group of size n (it is not known which t of the n users are corrupted). Column 'Conc.' indicates, whether the protocol allows for concurrent updates, column 'Rounds' the number of rounds required to recover from corruption, column 'Sender comm.' the cumulative uploaded communication, column 'Recipient comm.' the per-user download communication cost, and column 'Cost after rec.' the sender communication incurred by an update of a single user after the recovery process has concluded. TreeKEM I corresponds to the conservative approach of only healing by sending commits, TreeKEM II to using update proposals to heal at the expense of extra blanking. *: [BDR20] only achieves weak PCS, obtaining PCS guarantees similar to the rest would need $\mathcal{O}(n)$ cost after healing, due to extensive tainting.

To achieve FS, users should delete secret keys of outdated ratchet trees as soon as possible. For blockchains with immediate finality (i.e., no forks) this means old keys can be deleted immediately once a new ratchet tree is computed, while in longest-chain protocols one should wait to delete keys until the corresponding blocks are considered confirmed. Otherwise they might lose access to the group should a fork occur.

Efficiency. We now discuss the efficiency of DeCAF in healing a group with t compromises, and how it compares to related protocols. Throughout we refer to Table 6.1. There, we distinguish between two modes of TreeKEM (Propose and Commit). TreeKEM I corresponds to the conservative approach of only healing by sending commits (which would be expected behaviour, as argued below), hence is not concurrent. TreeKEM II, in turn corresponds to using update proposals to heal at the expense of extra blanking. Note that an execution where, as a rule, users achieve PCS by sending update proposals instead of commit is not compatible with retaining logarithmic communication in the long term, due to the large amount of blanks, as illustrated on the last column of Table 6.1. Thus, the data shown for the communication complexity of the latter mode of TreeKEM during healing is only short term. In order to have the fairest comparison, we consider the complexity of DeCAF in the decentralized setting and that of CoCoA in the centralized one, in which it was proposed.

We consider the process by which the group heals from t compromises. We first stress that since a party does not know if they are corrupted, they cannot decide whether to update based on this. The main novelty of our protocol is that the number of rounds that it takes to heal depends on the number of corrupted parties, but *not on the relative update behaviour of users*. Indeed, while several previous protocols could heal faster

than what is shown on the table in an optimal execution, this execution needs for the users and/or the server to coordinate and/or make "optimal" choices obliviously (since, again, there is no reason the identities of corrupted parties are known); for instance, give preference to the corrupted parties in the case of concurrency, or coordinate to not concurrently commit or update. In the table we consider thus all users updating. This is the case for TreeKEM I and Causal TreeKEM, who could heal optimally in t rounds, and thus reduce the communication complexity accordingly; but also for TreeKEM II, [BDR20] and [WKHB21a], for which the number of rounds is not affected, but whose communication complexity could be reduced in an optimal execution.

One can see that, among the protocols that provide sub-linear communication costs for sending updates over the long term, our protocol manages to heal in the least amount of rounds. On the recipient side, our protocol performs within a logarithmic factor of all others, except for CoCoA, which naturally outperforms all other in this regard, due to users only storing a partial view of the tree. We stress that, if run in the decentralized setting, CoCoA loses its advantage in terms of recipient communication, leading to a cost of $\mathcal{O}(n\log(n)^2)$. Thus, in this setting it is outperformed by DeCAF in every aspect.

6.1.2 Related Work

Sections 1.3 and 5.1.2 already introduced most relevant literature regarding (concurrent) secure (group) messaging. Here, we discuss some works regarding updatable encryption that are of particular relevance for this chapter, as well as examples of decentralized messaging apps.

The paper by Jost *et al.* [JMM19] first introduced the primitive of *secret key updatable public key encryption* (skuPKE) in the context of two-party messaging. In the context of group messaging, rTreeKEM [ACDT20] was the first protocol to formally use a (somewhat different to that of [JMM19]) version of updatable public key encryption, where decryption keys are updated upon being used. Nevertheless, though never formalized as a standalone primitive, a bit earlier Causal TreeKEM [Mat19] employed an enhanced version of PKE that can be seen as a modification of skuPKE where keys and update tokens have the same distribution. In our paper, we use the primitive and syntax from [JMM19], albeit with different security requirements. Many papers have proposed UPKE schemes with varying syntax and security definitions [DKW21, EJKM22, HLP22, AW23, HPS23]. Notably, a recent paper by Alwen, Fuchsbauer and Mularczyk [AFM24], enhanced prior UPKE notions to capture extra properties that are important for multi-party protocols, such as group messaging. In particular, it is the first to capture so-called forking security, i.e., security even allowing the adversary to update a key using different sequences of updates. The authors also propose an efficient pairing-based construction satisfying the stronger definition.

Last, some deployed messaging apps built around decentralization are Matrix [Fou], ToxChat[5], and Delta Chat[6] [SMP24]. The use of blockchain for CGKA protocols is novel as far as we know, but note that there exist previous messaging protocols making use of it, like Elixxir [Coi].

6.2 Preliminaries

6.2.1 Blockchain-aided Continuous Group-key Agreement

We now introduce blockchain-aided continuous group-key agreement (baCGKA), which allows the set up of a group $G = (\mathrm{ID}_1, \ldots . \mathrm{ID}_n)$ of users sharing an evolving group key. The definition is, naturally, similar to Definition 2.3.1, though it lacks an algorithm $\mathrm{CGKA.Dlv}$ for the server, since there is none in this setting. Additionally, it includes algorithms to explicitly send a previously crafted protocol message, and to fetch blocks of messages from the append-only data structure.

For the convenience of the reader, we describe the primitive below. Recall that we assume all users ID have an initialization key packet $((\mathrm{pk}_{\mathrm{ID}}, \mathrm{sk}_{\mathrm{ID}}), (\mathrm{svk}_{\mathrm{ID}}, \mathrm{ssk}_{\mathrm{ID}}))$, known to all other users.

A baCGKA scheme baCGKA specifies a tuple of algorithms $\mathrm{baCGKA.Init}$, $\mathrm{baCGKA.Upd}$, $\mathrm{baCGKA.Add}$, $\mathrm{baCGKA.Rem}$, $\mathrm{baCGKA.Proc}$, $\mathrm{baCGKA.Key}$, $\mathrm{baCGKA.Send}$, and finally $\mathrm{baCGKA.Fetch}$. The first six algorithms are local, in the sense that they only affect the executing user's state, and generate protocol messages to be sent to the rest of the group. The last two algorithms, on the other hand, interact with the distributed protocol by sending transactions and fetching blocks, respectively.

We consider a setting in which an append-only data structure is used to store the protocol messages and the data is distributed among several nodes. Users send their protocol messages to these nodes and then these nodes run a consensus algorithm that guarantees that they agree on their view of the data and on a total ordering of the blocks formed by the protocol messages. A blockchain is an example of this, and that is why we use the term "blockchain-aided" CGKA.

Initialization. User ID_1 runs $(\gamma, W) \leftarrow \mathrm{baCGKA.Init}(G, (\mathrm{pk}_{\mathrm{ID}_1}, \ldots, \mathrm{pk}_{\mathrm{ID}_n}), \mathrm{ssk}_{\mathrm{ID}_1})$ to initialize a session . Here $G = (\mathrm{ID}_1, \ldots, \mathrm{ID}_n)$ specifies the group, $\mathrm{pk}_{\mathrm{ID}_i}$ is the initialization encryption public-key of user ID_i, and $\mathrm{ssk}_{\mathrm{ID}_1}$ the initialization authentication secret key of the party setting up the group. The output consists of user ID_1's initial state and a welcome message W.

[5] https://tox.chat
[6] https://delta.chat/en/

Updates. To update their state, ID runs $(\gamma, U) \leftarrow \mathrm{baCGKA.Upd}(\gamma)$, updating their state and generating an update message.

Adding a group member. To add user ID' to the group member ID can run $(\gamma, A) \leftarrow \mathrm{baCGKA.Add}(\gamma, \mathrm{ID}', \mathrm{pk}_{\mathrm{ID}'})$. Here $\mathrm{pk}_{\mathrm{ID}'}$ is the initialization public key of ID' and A an add message.

Removing a group member. User ID can remove a (not necessarily different) user ID' from the group by running $(\gamma, R) \leftarrow \mathrm{CGKA.Rem}(\gamma, \mathrm{ID}')$. The output consists of an updated state and a removal message R.

Processing a block. To process a block B consisting of update, welcome, add, and remove messages, and move to an updated state, user ID runs $\gamma \leftarrow \mathrm{baCGKA.Proc}(\gamma, B)$.

Retrieving the group key. At any point a party ID in the group can extract the current group key K from its local state γ by running $K \leftarrow \mathrm{baCGKA.Key}(\gamma)$.

Sending a transaction. To send a transaction, i.e. a protocol message M generated by one of the previous algorithms, user ID runs algorithm $\mathrm{baCGKA.Send}(\gamma, M)$.

Fetch new blocks. Algorithm $(B_1, \ldots, B_\ell) \leftarrow \mathrm{baCGKA.Fetch}(\gamma)$ returns all blocks added to the chain since the user last fetched them.

6.2.2 Secretly Key-Updatable Public-Key Encryption

We now recall the definition of secretly key-updatable public-key encryption (skuPKE) schemes [JMM19]. A skuPKE scheme is essentially a public-key encryption scheme, that additionally allows the sampling of pairs (Δ, δ) of public and secret update information, which can be used to update secret and public keys, in a consistent way.

Defintion 6.2.1. *A* secretly key-updatable public-key encryption scheme kuPKE *consists of the tuple of algorithms* $(\mathrm{kuPKE.Gen}, \mathrm{kuPKE.Enc}, \mathrm{kuPKE.Dec}, \mathrm{kuPKE.Sam}, \mathrm{skuPKE.UpdP}, \mathrm{skuPKE.UpdS})$.
Key-generation algorithm $\mathrm{kuPKE.Gen}$ *on input of the security parameter* 1^λ *returns a key pair* $(\mathrm{pk}, \mathrm{sk})$. *Encryption algorithm* $\mathrm{kuPKE.Enc}$ *on input of public key* pk *and message* m *returns a ciphertext* c. *The deterministic decryption algorithm* $\mathrm{kuPKE.Dec}$ *receives as input a secret key* sk *and a ciphertext* c *and returns either a message* m *or the symbol* $\perp$ *indicating a decryption failure. Sampling algorithm* $\mathrm{kuPKE.Sam}(1^\lambda)$ *is used to sample pairs* (Δ, δ) *consisting of public and secret update information. The key-update algorithms* $\mathrm{skuPKE.UpdP}$ *and* $\mathrm{skuPKE.UpdS}$ *get as input* (pk, Δ) *and* (sk, δ)*, respectively, and output a rerandomized key* pk' *or* sk'.

Correctness essentially requires that updating the public and secret key of a key-pair with the same sequence of rerandomization factors preserves compatibility of the updated keys with each other. More precisely let $\lambda, k \in \mathbb{N}$, and each pair $(\mathrm{pk}_0, \mathrm{sk}_0) \in [\mathrm{kuPKE.Gen}(1^\lambda)]$, and $(\Delta_0, \ldots, \Delta_k)$, $(\delta_0, \ldots, \delta_k)$ be vectors with $(\Delta_i, \delta_i) \in [\mathrm{kuPKE.Sam}(1^\lambda)]$ for all i. Further, for $i \in \{0, \ldots, k\}$ let $\mathrm{pk}_{i+1} = \mathrm{skuPKE.UpdP}(\mathrm{pk}_i, \Delta_i)$ and $\mathrm{sk}_{i+1} = \mathrm{skuPKE.UpdS}(\mathrm{sk}_i, \delta_i)$. We require that for all messages m and all i, $\mathrm{PKE.Dec}(\mathrm{sk}_i, \mathrm{PKE.Enc}(\mathrm{pk}_i, m)) = m$.

Security. For security we essentially require that, on one hand, messages encrypted to a secret key that was generated by updating a potentially compromised secret key are secure as long as the secret update information to do so was not leaked, and, on the other hand, that leaking an updated key does not compromise ciphertexts encrypted to its predecessor as long as the secret update information was not leaked. More precisely, we say that kuPKE is *secure* with respect to an upper bound L on the number of key updates, if it satisfies the following security guarantees:

Defintion 6.2.2. *Let* $(\mathrm{pk}_0, \mathrm{sk}_0) \leftarrow \mathrm{kuPKE.Gen}(1^\lambda)$ *and also let* $(\Delta_0, \ldots, \Delta_{Q-1})$, $(\delta_0, \ldots, \delta_{Q-1})$ *with* $(\Delta_i, \delta_i) \leftarrow \mathrm{kuPKE.Sam}(1^\lambda)$, *and let* s *and* s_i *denote the random coins used by* $\mathrm{kuPKE.Gen}$ *and* $\mathrm{kuPKE.Sam}$, *respectively. For* $i \in [Q-1]_0$ *define* $\mathrm{pk}_{i+1} = \mathrm{skuPKE.UpdP}(\mathrm{pk}_i, \Delta_i)$ *and* $\mathrm{sk}_{i+1} = \mathrm{skuPKE.UpdS}(\mathrm{sk}_i, \delta_i)$. *Then,* kuPKE *is* IND-CPA *secure, if for any choice* ρ, j^-, j^+ *with* $-1 \leq j^- < \rho \leq j^+ \leq Q$ *and messages* m_0, m_1 *it holds that*

$$\mathrm{kuPKE.Enc}(\mathrm{pk}_\rho, m_0) \approx_c \mathrm{kuPKE.Enc}(\mathrm{pk}_\rho, m_1) \,,$$

even given access to $(\mathrm{pk}_i)_{i \in [L]_0}$, $(\mathrm{sk}_i)_{i \in [Q]_0 \setminus [j^- + 1, j^+]}$, $(\Delta_i)_{i \in [Q-1]_0}$, $(\delta_i)_{i \in [Q-1]_0 \setminus \{j^-, j^+\}}$, *as well as random coins* s *if* $j^- \geq 0$, *and* $(s_i)_{i \in [Q-1]_0 \setminus \{j^-, j^+\}}$.

Our variant of IND-CPA is incomparable to the one required for two party ratcheting [JMM19]; in this work the update information can be generated using adversarially chosen randomness, and the challenge ciphertext encrypts a message, that contains secret update information, giving the security notion a circular flavor. On the other hand, only one secret key is ever exposed to the adversary, while in our notion several are. Compared to [ACDT20] our security notion is stronger; in this work the authors use kuPKE mainly to achieve improved forward secrecy. Accordingly, their variant of IND-CPA roughly requires that access to updated secret keys does not allow to compromise encryption to previous keys, as long as the update information used to generate the corrupted key remains secure.

Instantiations. A very efficient instantiation of skuPKE can be constructed in prime order groups $(\mathbb{G}, g, p)$. The scheme is essentially the Hashed ElGamal scheme [ABR01],

where update information is of the form $(\Delta = g^\delta, \delta)$ with $\delta \in \mathbb{Z}_p$ uniformly random, and key pairs $(X = g^x, x)$ are updated as $x + \delta$ and $X \cdot \Delta$ respectively. For standard-model instantiation see [DKW21, HPS23].

Definition of the scheme. The key-generation, encryption and decryption algorithms work as in the Hashed ElGamal scheme. That is, $\mathrm{kuPKE.Gen}(1^\lambda)$ outputs a pair $(\mathrm{pk}, \mathrm{sk}) = ((\mathbb{G}, p, g, g^x, \mathsf{H}), (\mathbb{G}, p, g, x, \mathsf{H}))$, where $\mathbb{G}$ is a group of prime order p (the bit length of p is λ), g is a generator of $\mathbb{G}$, x is sampled at random from $\mathbb{Z}_p$ and H is a hash function that takes elements in $\mathbb{G}$ as input and outputs strings in $\{0,1\}^\lambda$. An encryption of a message $m \in \{0,1\}^\lambda$ using the public key g^x is a pair $(g^y, \mathsf{H}((g^x)^y) \oplus m)$ where y is sampled at random from $\mathbb{Z}_p$. The decryption algorithm takes as input a ciphertext (c_1, c_2) and a private key x and outputs $\mathsf{H}((c_1)^x) \oplus c_2$.

The sampling algorithm $\mathrm{kuPKE.Sam}(1^\lambda)$ outputs a pair $(\Delta = g^\delta, \delta)$ where δ is sampled from the uniform distribution over $\mathbb{Z}_p$. Public-key-update algorithm $\mathrm{skuPKE.UpdP}$ gets as input (g^x, Δ) and outputs $g^x\Delta$, while $\mathrm{skuPKE.UpdS}$ takes (x, δ) as input and outputs $x + \delta$.

The security proof is based on a standard IND-CPA security proof of Hashed ElGamal like the one that can be found on textbooks and it is provided for completeness. It relies on the hardness of the computational Diffie-Hellman (CDH) problem and uses the random oracle model.

We say that the CDH problem is hard with respect to $\mathrm{kuPKE.Gen}$ if for every PPT algorithm A there exists a negligible function $\varepsilon(n)$ such that

$$\Pr[\mathsf{A}(\mathbb{G}, q, g, g^x, g^y) = g^{xy}] \leq \varepsilon(n),$$

where the probabilities are taking over the randomness used by $\mathrm{kuPKE.Gen}$ to generate $(\mathbb{G}, q, g)$ and x and y are sampled uniformly from $\mathbb{G}$.

Theorem 6.2.3. *If the CDH problem is hard with respect to* $\mathrm{kuPKE.Gen}$ *and* H *is modeled as a random oracle, the Hashed ElGamal skuPKE scheme is IND-CPA secure.*

Proof. Let ρ, j^-, j^+ be a set of indices such that $-1 \leq j^- < \rho \leq j^+ \leq L$ and A be a PPT adversary trying to distinguish

$$\mathrm{kuPKE.Enc}(\mathrm{pk}_\rho, m_0) \approx_c \mathrm{kuPKE.Enc}(\mathrm{pk}_\rho, m_1)$$

as in Definition 6.2.2.

Let $(g^y, \mathsf{H}((\mathrm{pk}_\rho)^y) \oplus m_b)$ denote a ciphertext. As the hash function is modeled as a random oracle, A cannot distinguish the ciphertexts with probability greater than $1/2$ unless it makes a query to the random oracle on $(\mathrm{pk}_\rho)^y$. Let E denote the event that

such a query is made. Therefore the probability that A is able to distinguish the two distributions is bounded by $1/2 + \Pr[E]$.

We now show that $\Pr[E]$ is negligible. We define an algorithm B that takes as input a CDH challenge $(\mathbb{G}, p, g, g^x, g^y)$ and uses A as a subroutine. It samples $b \leftarrow \{0,1\}$ and $(\Delta_i, \delta_i) \leftarrow \mathrm{kuPKE.Sam}(1^\lambda)$ for $i \in \{0, \ldots, j^- - 1\} \cup \{j^- + 1, \ldots, j^+ - 1\} \cup \{j^+ + 1, \ldots, L-1\}$. It chooses g^x as the ρ-th public key, $(\mathrm{pk}_{j^-}, \mathrm{sk}_{j^-}) = (g^{r^-}, r^-)$ and $(\mathrm{pk}_{j^+ +1}, \mathrm{sk}_{j^+ +1}) = (g^{r^+}, r^+)$ where r^-, r^+ are uniformly chosen in $\mathbb{Z}_p$. It computes $\Delta_{j^-} = g^x (\prod_{i=j^-+1}^{\rho-1} \Delta_i)^{-1} g^{-r^-}$ and $\Delta_{j^+} = g^{-x} (\prod_{i=\rho}^{j^+-1} \Delta_i)^{-1} g^{r^+}$. The remaining public and private keys are chosen accordingly, that is,

$$
\begin{aligned}
\mathrm{pk}_i &= \mathrm{pk}_{i+1} \cdot \Delta_i^{-1} && \text{for } i \in \{\rho - 1, \ldots, 0\} \\
\mathrm{pk}_i &= \mathrm{pk}_{i-1} \cdot \Delta_{i-1} && \text{for } i \in \{\rho + 1, \ldots, L\} \\
\mathrm{sk}_i &= \mathrm{sk}_{i+1} - \delta_i && \text{for } i \in \{j^- - 1, \ldots, L\} \\
\mathrm{sk}_i &= \mathrm{sk}_{i-1} + \delta_{i-1} && \text{for } i \in \{j^+ + 2, \ldots, L\}
\end{aligned}
$$

Then B sends to A $(\mathrm{pk}_i)_{i \in [L]_0}$, $(\mathrm{sk}_i)_{i \in [L]_0 \setminus [j^- + 1, j^+]}$, $(\Delta_i)_{i \in [L-1]_0}$, $(\delta_i)_{i \in [L-1]_0 \setminus \{j^-, j^+\}}$ as well as the random coins used by $\mathrm{kuPKE.Gen}$ and $\mathrm{kuPKE.Sam}$ as specified in Definition 6.2.2.

As an observation, B can actually compute those secret keys because it first chooses sk_{j^-} and $\mathrm{sk}_{j^+ +1}$, and then it proceeds recursively using the δ_i that it sampled before. The construction also guarantees that the pairs $(\mathrm{pk}_i, \mathrm{sk}_i)$ satisfy $g^{\mathrm{sk}_i} = \mathrm{pk}_i$.

When A makes a random oracle query $u \in \mathbb{G}$, B sends a random string s_u and keeps a list of pairs (u, s_u). When A sends two messages m_0, m_1, B replies with a ciphertext $(g^y, k \oplus m_b)$ where k is sampled uniformly at random.

Finally, B chooses a random pair in the list of random oracle queries A made and outputs the first component.

Since the view of A as an IND-CPA adversary and when run as a subroutine of B before it makes a query to the random oracle on $(\mathrm{pk}_\rho)^y$ is the same, the probability that E happens is the same in both cases. This is because if A does not make said query then B perfectly simulates the IND-CPA game. Let Q denote the number of random oracle queries. By construction, when A is run as a subroutine of B, $\Pr[E]/Q \leq \Pr[\mathsf{B}(\mathbb{G}, q, g, g^x, g^y) = g^{xy}] \leq \varepsilon(\lambda)$ for some negligible function by hypothesis. Hence the probability that A is able to distinguish the two distributions is bounded by $1/2 + Q \cdot \varepsilon(\lambda)$, i.e., the Hashed ElGamal skuPKE scheme is IND-CPA secure. □ □

6.3 Protocol description

We now describe DeCAF in detail. Section 6.3.1 describes how the protocol proceeds in epochs determined by the blockchain's blocks, Section 6.3.2 describes the contents of a user's state, Section 6.3.3 how the structure of the ratchet tree is modified when handling changes to the group membership, and Section 6.3.4 how update information for a path in the ratchet tree is samped and applied. Finally, in Section 6.3.5 we give the formal description of the protocol's algorithms.

6.3.1 Blocks and Epochs

DeCAF proceeds in epochs consisting of k blocks. More precisely the ith epoch corresponds to blocks $i+1, \ldots, i+k$ of the blockchain. Updates are generated with respect to the ratchet tree of the *first* block of the current epoch. This is to handle potential delays of up to k blocks from the moment a user sends a message containing group operations information to the moment it makes it into the blockchain. At the beginning of a new epoch, the group switches to a new ratchet tree that incorporates all updates of the last epochs, as well as the dynamic changes made to the group. One consequence of having to accommodate for such delays is that users need to store at least the keys at the beginning of an epoch for the entire duration of it, and if the underlying blockchain does not have immediate finality potentially keys from further back. This translates into weaker FS guarantees than in the server setting as a user cannot immediately delete keys after updating to the next state. But this difference will be marginal as the length of an epoch (or confirmation time of the blockchain, whichever is larger) will still be tiny compared to the duration for which users are typically offline. A second consequence is that these delays introduce a further delay in the execution of dynamic operations. Indeed, updating information generated during an epoch is computed without taking into account users that were being removed or added during that round. Thus, in the case of epochs with adds, the key at the end of that epoch will not be known to the new parties, who will need to wait another round to learn it. In the case of epochs with removes, the key at the end of that epoch will be blank, so a new key will be necessary to establish a new group key that the removed users do not have knowledge of. We remark that this seems to be somewhat inherent. In fact, if we set $k = 1$, the situation is not that different than that in other protocols like CoCoA or TreeKEM, where a first round of dynamic operations needs to be followed by a subsequent one where the commit effecting the operations takes place. In summary, using a blockchain for decentralization gives improved consistency and security guarantees, but the delay between protocol rounds is now dictated by the block arrival and typical inclusion times of the underlying blockchain. Therefore, FS is (marginally) affected by the confirmation time of blocks.

ID	user identifier
$\mathcal{T}$	ratchet tree at the beginning of the current epoch
$\mathcal{T}_{\mathrm{next}}$	working copy of the ratchet tree for the next epoch
O_{next}	dynamic operations to be implemented before next epoch
U_{pend}	pending update
e_{ctr}	epoch counter
I	the epoch's group key
I_{next}	working copy of next epoch's group key
$(\mathrm{pk}_c, \mathrm{sk}_c)$	dummy key pair

Table 6.2: User ID's state.

6.3.2 Users' States

User ID's state $\mathrm{ID}.\gamma$ contains the user's identifier ID, two ratchet trees $\mathcal{T} = (V, E)$ and $\mathcal{T}_{\mathrm{next}} = (V_{\mathrm{next}}, E_{\mathrm{next}})$, lists O_{next}, and U_{pend}, epoch counter e_{ctr}, a key pair $(\mathrm{pk}_c, \mathrm{sk}_c)$, the (potentially empty) group key I, and a working copy of the group key I_{next} for the next epoch. For an overview see Table 6.2.

$\mathcal{T}$ contains the state of the ratchet tree at the beginning of the current epoch. More precisely, this encompasses the public states ${}^{p}\gamma(v)$ of all nodes $v \in V$ and, if we denote ID's leaf in $\mathcal{T}$ by v_{ID}, additionally the secret node states for all nodes v in ID's update path $\mathrm{path}(v_{\mathrm{ID}})$. Ratchet tree $\mathcal{T}_{\mathrm{next}}$ serves as a working copy for the next epoch, i.e., it contains keys updated according to the blocks already processed in the current epoch—excluding dynamic operations. Note that the two trees differ only in the node states, but not the general tree structure. To clarify whether we consider nodes in $\mathcal{T}$ or $\mathcal{T}_{\mathrm{next}}$, we will denote nodes in the latter by v^{n}. O_{next} is a list of the dynamic operations included in the blocks of the current epoch that were already processed. These changes will be applied to $\mathcal{T}_{\mathrm{next}}$ at the end of the epoch. List U_{pend} stores pending update information. The epoch counter e_{ctr} is used to generate and confirm protocol messages for the current epoch. Finally $(\mathrm{pk}_c, \mathrm{sk}_c)$ is the dummy key-pair used for blank nodes.

6.3.3 Implementing Dynamic Operations

As a result of dynamic operations, the tree structure will change. Here, we describe this change, ahead of the protocol description.

To add parties we use an adaptation of the *unmerged leaves* technique (see Section 2.6) introduced in TreeKEM v9[BBR+23]. In particular, whenever ID, whose path contains a node where another party ID' is unmerged, generates an update, they need to encrypt the current key for that node, together with the seed used to sample the update information to ID'. However, this key might already have been present in an epoch

which preceded that in which ID' was added. Hence, sending it to ID could cause problems with forward secrecy—ID must ensure that the key sent to ID' was updated *after* they joined the group. Thus, this process is done in two steps. First, upon being added to the group, ID' is included into the set $v.unm_0$ for all v in their path, except for the root. Updates that apply to v, issued while ID' is in this set $v.unm_0$, do not encrypt any secret information about v to ID. Whenever an epoch first contains such an update for v, however, ID' is removed from the set $v.unm_0$ and added to $v.unm_1$, at the end of the epoch. This signals that the key at v is now safe to be communicated to ID'. Any following update that applies to v once $\mathrm{ID}' \in v.unm_1$, will then encrypt the current key plus the update information to ID'. Once such an update occurs, ID' learns the key at v, and is then removed from $v.unm_1$. The one exception to this is the root node v_{root}, where ID' is directly added to $v_{root}.unm_1$. The reason for this is that all add operations are coupled with an update from the issuing party, thus ensuring that the root key at the end of that epoch is updated, and thus safe to communicate to ID'.

Removes are handled via *blanking*, where the keys that removed users had knowledge of get set to the dummy key-pair $(\mathrm{pk}_c, \mathrm{sk}_c)$ and get ignored by users encrypting new secret update information δ_i until they get updated again in a subsequent epoch.

All these changes are executed once at the end of each epoch. While all group operations in the following epoch will take the new tree into account, added and removed users will not be properly added and removed until the end of that following epoch, though. This seems inherent if we want to allow concurrency: the author of an operation concurrent with a dynamic one will be oblivious to the latter, thus unable to prepare their operation taking it into account.

More in detail, at the end of an epoch where adds $A = (A_1, \ldots, A_{\ell_a})$, removes $R = (R_1, \ldots, R_{\ell_r})$, and modifications $M = (M_1, \ldots, M_{\ell_m})$ to the sets of unmerged users took place, users will call algorithm $\mathrm{upd\text{-}tree}(\mathcal{T}_{\mathrm{next}}, A, R, M)$, which will output the tree resulting from applying these operations. First, the algorithm in order processes the M_i, which are lists of nodes that were affected by updates in the current epoch (their exact definition is given in Section 6.3.4 below). For every $v \in M$ the sets of unmerged leaves are updated to $v.unm_1 \leftarrow v.unm_0$ and $v.unm_0 \leftarrow \emptyset$. Then, the algorithm will set the state of all in the paths of any of the removed users to *blank*, and associate with them the dummy key-pair $(\mathrm{pk}_c, \mathrm{sk}_c)$. Added parties will get assigned a leaf in the tree in a canonical way, determined by the ordering of operations in the corresponding block. The first leaves to be assigned will be blank ones, and new leaves to the right of the existing ones will be added, if there are not enough blanked ones, adding any internal nodes necessary to maintain the binary structure of the tree. If a new root node must be added to accommodate for the new parties, this will be given the dummy key-pair until it gets updated at the end of the next epoch. Then, for each newly-added party ID_i with init key $\mathrm{pk}_{\mathrm{ID}}$, it sets the state of their new leaf v_{ID} to

$(\mathrm{pk}_{\mathrm{ID}}, \mathrm{svk}_{\mathrm{ID}})$, and for any $v \in \mathrm{path}(l_i)$ except the root v_{root}, it adds ID_i to $v.unm_0$. The root id_i is added to $v_{root}.unm_1$. Finally, it outputs the resulting tree.

Whenever an update including new update information for a node v takes place, v will become unblanked if it was not so already, as expected. Moreover, unmerged leaves in unm_1 will become merged, and those in unm_0 will then pass to unm_1.

6.3.4 Updating the States of an Update Path

During the initialization of a group and when updating, users will update the keys along some path. Before turning to the description of our protocol's algorithms, we detail this operation.

Consider user ID with associated leaf v_{ID}. Update information for the keys of $\mathrm{path}(v_{\mathrm{ID}})$ is sampled using

$$((\Delta_i, \delta_i, C_i)_i, \kappa) \leftarrow \text{gen-path-upd}(\gamma) \ .$$

The algorithm, on input of the user's state, first fetches $(v_1 = v_{\mathrm{ID}}, \ldots, v_r = v_{root}) = \mathrm{path}(v_{\mathrm{ID}})$ with respect to ratchet tree $\mathcal{T}$ corresponding to the beginning of the epoch. Let m be maximal such that $\mathrm{ID} \in v_{m-1}.unm_0 \cup v_{m-1}.unm_1$. If no such m exists, we set $m = 2$. The algorithm samples a seed s_1 uniformly at random and computes $s_m = \mathsf{H}_1(s_1)$ as well as $s_i = \mathsf{H}_1(s_{i-1})$ for $i = m+1, \ldots, r$. For $i \in \{1, m, \ldots, r\}$ it samples update information $(\Delta_i, \delta_i) \leftarrow \mathrm{kuPKE.Sam}(\mathsf{H}_2(s_i))$ using randomness $\mathsf{H}_2(s_i)$. It then for $i \in \{m, \ldots, r\}$ computes vectors of ciphertexts $C_i = (c_{i,j})_{z_j}$ with $c_{i,j} \leftarrow \mathrm{kuPKE.Enc}(z_j.\mathrm{pk}, s_i)$, where the nodes z_j are chosen as

$$z_j \in \mathrm{Res}(v_{i-1}) \cup v_i.unm_1 \setminus v_{i-1}.unm_1$$

for $i = m+1, \ldots, r$ and

$$z_j \in \mathrm{Res}(\mathrm{lparent}(v_i)) \cup \mathrm{Res}(\mathrm{rparent}(v_i)) \cup v_i.unm_1 \setminus \{\mathrm{ID}\}$$

for $i = m$. Finally, $\kappa = \mathsf{H}_1(s_r)$ will be used to update the group key. The algorithm's output is $((\Delta_i, \delta_i, C_i)_i, \kappa)$. Looking ahead, $(\Delta_i, C_i)_i$ will be sent out as the update message and $((\Delta_i, \delta_i)_i, \kappa)$ saved in the user's pending state.

When user ID' wants to apply a path update $(\Delta_i, C_i)_i$ with $i \in \{1, m, \ldots, r\}$ generated by user ID, they call algorithm

$$\mathrm{ID}'.\gamma \leftarrow \text{proc-path-upd}(\mathrm{ID}'.\gamma, (\Delta_i, C_i)_i) \ .$$

It first fetches user ID's update path $(v_1^{\mathrm{n}} = v_{\mathrm{ID}}^{\mathrm{n}}, \ldots, v_r^{\mathrm{n}} = v_{root}^{\mathrm{n}}) = \mathrm{path}(v_{\mathrm{ID}}^{\mathrm{n}})$ from the working copy $\mathcal{T}_{\mathrm{next}}$ of the ratchet tree. Then, for all i it updates the public keys along the path, i.e., $v_i^{\mathrm{n}}.\mathrm{pk} \leftarrow \mathrm{skuPKE.UpdP}(v_i^{\mathrm{n}}.\mathrm{pk}, \Delta_i)$. Here, if v_i^{n} was blank and thus

has no associated public key, the public key of a constant dummy key-pair $(\mathrm{pk}_c, \mathrm{sk}_c)$ is used as $v_i^{\mathrm{n}}.\mathrm{pk}$. Note that this implies that v_i^{n}'s resolution is now $\{v_i^{\mathrm{n}}\}$.

Let v_i denote the first node that is shared between $\mathrm{path}(v_{\mathrm{ID}})$ and $\mathrm{path}(v_{\mathrm{ID}'})$ and for which $\mathrm{ID}' \notin v_i.unm_0$. Then, if the update was generated during the current epoch, C_i contains an encryption $c_{i,j}$ of seed s_i under the public key of some node $w_{i,j}$ for which the secret key is contained in ID's copy of tree $\mathcal{T}$ that is part of $v_{\mathrm{ID}'}.\gamma$. The algorithm recovers $s_i \leftarrow \mathrm{kuPKE.Dec}(w_{i,j}.\mathrm{sk}, c_{i,j})$ and for $j \in \{i+1, \ldots, r\}$ computes $s_j = \mathsf{H}_1(s_{j-1})$ and update information $(\Delta_j, \delta_j) \leftarrow \mathrm{kuPKE.Gen}(\mathsf{H}_2(s_j))$. It then updates the corresponding secret keys in $\mathcal{T}_{\mathrm{next}}$ as $v_j^{\mathrm{n}}.\mathrm{sk} \leftarrow \mathrm{skuPKE.UpdS}(v_j^{\mathrm{n}}.\mathrm{sk}, \delta_j)$, where, analogous to the above, if v_j is blank, sk_c takes the role of $v_j.\mathrm{sk}$. Finally, the algorithm computes group key update information $\kappa = \mathsf{H}_1(s_r)$, incorporates it in the working copy of the group key $I_{\mathrm{next}} \leftarrow I_{\mathrm{next}} \oplus \kappa$, and adds the list $M = (v_m, \ldots, v_r)$ to O_{next}. The latter will be used to update the sets of unmerged users at the end of the epoch.

6.3.5 Protocol Algorithms

Initialization. To initialize a group for users $(\mathrm{ID}_1, \ldots, \mathrm{ID}_n)$, user ID_1 first generates the dummy key-pair $(\mathrm{pk}_c, \mathrm{sk}_c) \leftarrow \mathrm{kuPKE.Gen}(1^\lambda)$. They then set up a left-balanced binary ratchet tree $T = (V, E)$. Every node in $\mathcal{T}$ is blank, except for the leaves. The public state of the i^{th} leaf contains the corresponding user's initialization public key and their signature verification key. Further, the secrets state of ID_1, contains ID_1's secret decryption and signing keys. Group creator ID_1 incorporates $(\mathrm{pk}_c, \mathrm{sk}_c)$, $\mathcal{T}$, a copy $\mathcal{T}_{\mathrm{next}}$ of $\mathcal{T}$, and an empty list O_{next} in their state and computes $((\Delta_i, \delta_i, C_i)_i, \kappa) \leftarrow \mathrm{gen\text{-}path\text{-}upd}(\mathrm{ID}_1.\gamma)$. The tuple $((\Delta_i, \delta_i)_i, \kappa)$ is added to ID_1's state together with epoch counter $e_{\mathrm{ctr}} = (1, 1)$ (where the first coordinate denotes the epoch and the second one the block inside the epoch) and $I_{\mathrm{next}} \leftarrow 0$. The algorithm outputs the resulting state and welcome message $W = ({}^p\mathcal{T}, (\Delta_i, C_i)_i, (\mathrm{pk}_c, \mathrm{sk}_c), \sigma, \mathrm{ID}_1)$, where σ is a signature of $({}^p\mathcal{T}, (\Delta_i, C_i)_i, (\mathrm{pk}_c, \mathrm{sk}_c))$ under $\mathrm{ssk}_{\mathrm{ID}_1}$.

Update. To issue an update, ID computes $((\Delta_i, \delta_i, C_i)_i, \kappa) \leftarrow \mathrm{gen\text{-}path\text{-}upd}(\mathrm{ID}.\gamma)$. The secret update information $(\delta_i)_i$ and κ are stored in ID's pending state U_{pend}. Let $(v_1, \ldots, v_r) = \mathrm{path}(v_{\mathrm{ID}})$ be ID's update path. Update messages also communicate the current secret key of nodes to unmerged users that have already processed an update on this node. More precisely, the updating user for all $i \in [2, \ldots, r]$ such that $\mathrm{ID} \notin v_i.unm_0 \cup v_i.unm_1$ computes a vector of ciphertexts $\tilde{C}_i = (\tilde{c}_{i,j})_{z_j}$, where $\tilde{c}_{i,j} = \mathrm{kuPKE.Enc}(z_j.\mathrm{pk}, v_i.\mathrm{sk})$ and z_j are the nodes satisfying $z_j \in v_i.unm_1$. The algorithm outputs message $U = ((\Delta_i, C_i)_i, (\tilde{C})_i, e_{\mathrm{ctr}}, \sigma, \mathrm{ID})$, where σ is a signature of $((\Delta_i, C_i)_i, (\tilde{C})_i, e_{\mathrm{ctr}})$ under $\mathrm{ssk}_{\mathrm{ID}}$.

Alg DeCAF.Init$(G, (\mathrm{ID}_1, \mathrm{sk}, \mathrm{ssk}), (\mathrm{ID}_1, \ldots, \mathrm{ID}_n))$

00 $(\mathrm{pk}_c, \mathrm{sk}_c) \leftarrow \mathrm{kuPKE.Gen}(1^\lambda)$
01 $\mathcal{T} \leftarrow \text{gen-tree}(\mathrm{ID}_1, \ldots, \mathrm{ID}_n)$
02 $(v_{\mathrm{ID}}.\mathrm{sk}, v_{\mathrm{ID}}.\mathrm{ssk}) \leftarrow (\mathrm{sk}, \mathrm{ssk})$
03 $\gamma \leftarrow (\mathrm{ID}_1, \mathcal{T}, \mathcal{T}, \emptyset, \emptyset, (0,0), (\mathrm{pk}_c, \mathrm{sk}_c), 0, 0))$
04 $((\Delta_i, \delta_i, C_i)_{i\in(1,\ldots,r)}, \kappa) \leftarrow \text{gen-path-upd}(\gamma)$
05 $(v_1, \ldots, v_r) \leftarrow \mathrm{path}(v_{\mathrm{ID}}^T)$
06 **For** $i \in [r]$**:**
07 $\quad v_i.\mathrm{pk} \leftarrow \mathrm{skuPKE.UpdP}(\mathrm{pk}_c, \Delta_i)$
08 $\quad v_i.\mathrm{sk} \leftarrow \mathrm{skuPKE.UpdS}(\mathrm{sk}_c, \delta_i)$
09 $I_{\text{next}} \leftarrow \kappa$
10 $K \leftarrow \mathsf{H}_1(\text{`}\mathtt{key}', I_{\text{next}})$
11 $I_{\text{next}} \leftarrow \mathsf{H}_1(\text{`}\mathtt{next}', I_{\text{next}})$
12 $\gamma \leftarrow (\mathrm{ID}, \mathcal{T}, \mathcal{T}, \emptyset, \emptyset, (0,0), (\mathrm{pk}_c, \mathrm{sk}_c), K, I_{\text{next}})$
13 $\sigma \leftarrow \mathsf{Sig}(\mathrm{ssk}, ({}^p\mathcal{T}, (\Delta_i, C_i)_{i\in(1,\ldots,r)}, (\mathrm{pk}_c, \mathrm{sk}_c)))$
14 $W \leftarrow ({}^p\mathcal{T}, (\Delta_i, C_i)_{i\in(1,\ldots,r)}, (\mathrm{pk}_c, \mathrm{sk}_c), \sigma, \mathrm{ID}_1)$
15 **Return** (γ, W)

Alg DeCAF.Add(γ, ID')

16 $(\mathrm{ID}, \mathcal{T}, \mathcal{T}_{\text{next}}, O_{\text{next}}, U_{\text{pend}}, e_{\text{ctr}}, (\mathrm{pk}_c, \mathrm{sk}_c), K, I_{\text{next}}) \leftarrow \gamma$
17 $(\gamma, U) \leftarrow \mathrm{DeCAF.Upd}(\gamma)$
18 $\sigma \leftarrow \mathsf{Sig}(\mathrm{ssk}_{\mathrm{ID}}, (\text{`}\mathtt{add}(\mathrm{ID}')', {}^p\mathcal{T}, (\mathrm{pk}_c, \mathrm{sk}_c), U, e_{\text{ctr}}))$
19 $A \leftarrow (\text{`}\mathtt{add}(\mathrm{ID}')', {}^p\mathcal{T}, (\mathrm{pk}_c, \mathrm{sk}_c), U, e_{\text{ctr}}, \sigma, \mathrm{ID})$
20 **Return** (γ, A)

Alg DeCAF.Upd(γ)

21 $(\mathrm{ID}, \mathcal{T}, \mathcal{T}_{\text{next}}, O_{\text{next}}, U_{\text{pend}}, e_{\text{ctr}}, (\mathrm{pk}_c, \mathrm{sk}_c), K, I_{\text{next}}) \leftarrow \gamma$
22 $((\Delta_i, \delta_i, C_i)_{i\in(1,m,\ldots,r)}, \kappa) \leftarrow \text{gen-path-upd}(\gamma)$
23 $U_{\text{pend}} \leftarrow ((\delta_i)_{i\in(1,m,\ldots,r)}, \kappa)$
24 $(v_1, \ldots, v_r) \leftarrow \mathrm{path}(v_{\mathrm{ID}})$
25 \\ **Encryptions to unmerged users**
26 **For** $i \in (m, \ldots, r)$**:**
27 $\quad$ **If** $\mathrm{ID} \notin v_i.unm_0 \cup v_i.unm_1$**:**
28 $\quad\quad \tilde{C}_i \leftarrow \emptyset$
29 $\quad$ **For** $z \in v_i.unm_1$**:**
30 $\quad\quad$ **If** $v_i = v_{root}$**:**
31 $\quad\quad\quad \tilde{C}_i \xleftarrow{\cup} \mathrm{kuPKE.Enc}(z.\mathrm{pk}, I_{\text{next}})$
32 $\quad\quad$ **Else:**
33 $\quad\quad\quad \tilde{C}_i \xleftarrow{\cup} \mathrm{kuPKE.Enc}(z.\mathrm{pk}, v_i.\mathrm{sk})$
34 $\sigma \leftarrow \mathsf{Sig}(\mathrm{ssk}_{\mathrm{ID}}, ((\Delta_i, C_i)_{i\in(1,m,\ldots,r)}, (\tilde{C}_i)_{i\in(m,\ldots,r)}, e_{\text{ctr}})$
35 $U \leftarrow ((\Delta_i, C_i)_{i\in(1,m,\ldots,r)}, (\tilde{C}_i)_{i\in(m,\ldots,r)}, e_{\text{ctr}}, \sigma, \mathrm{ID})$
36 **Return** (γ, U)

Alg DeCAF.Rem(γ, ID')

37 $(\mathrm{ID}, \mathcal{T}, \mathcal{T}_{\text{next}}, O_{\text{next}}, U_{\text{pend}}, e_{\text{ctr}}, (\mathrm{pk}_c, \mathrm{sk}_c), K, I_{\text{next}}) \leftarrow \gamma$
38 $\sigma \leftarrow \mathsf{Sig}(\mathrm{ssk}_{\mathrm{ID}}, (\text{`}\mathtt{remove}(\mathrm{ID}')', e_{\text{ctr}}))$
39 $R \leftarrow (\text{`}\mathtt{remove}(\mathrm{ID}')', e_{\text{ctr}}, \sigma, \mathrm{ID})$
40 **Return** (γ, R)

Alg DeCAF.Key(γ)

41 $(\mathrm{ID}, \mathcal{T}, \mathcal{T}_{\text{next}}, O_{\text{next}}, U_{\text{pend}}, e_{\text{ctr}}, (\mathrm{pk}_c, \mathrm{sk}_c), K, I_{\text{next}}) \leftarrow \gamma$
42 **Return** K

Figure 6.3: DeCAF Algorithms for initializing the group, generating updates, adding and removing users. Algorithm gen-tree takes as input a list of user identifiers and outputs the ratchet tree with leaves having public state given by the identifiers and corresponding public keys. They employ the helper functions detailed in Fig. 6.5. For the algorithm that describes how to process the operations see Fig. 6.4.

Alg DeCAF.Proc(γ, B)

00 $(W, U, A, R) \leftarrow B$
01 \\ **Case:** ID **is already part of the group**
02 **If** $\gamma \neq (\mathrm{ID}, \mathrm{sk}, \mathrm{ssk})$**:**
03 $(\mathrm{ID}, \mathcal{T}, \mathcal{T}_{\mathrm{next}}, O_{\mathrm{next}}, U_{\mathrm{pend}}, e_{\mathrm{ctr}}, (\mathrm{pk}_c, \mathrm{sk}_c), K, I_{\mathrm{next}}) \leftarrow \gamma$
04 **For** $U_i \in U$**:**
05 $((\Delta_i, C_i)_{i\in(1,m,\ldots,r)}, (\tilde{C}_i)_{i\in(m,\ldots,r)}, e_{\mathrm{ctr}}^{\sim}, \sigma, \tilde{\mathrm{ID}}) \leftarrow U_i$
06 **Require** $\mathsf{Verify}(\mathrm{svk}_{\tilde{\mathrm{ID}}}, \sigma) = 1 \wedge e_{\mathrm{ctr}}^{\sim}[0] = e_{\mathrm{ctr}}[0]$
07 **If** $\tilde{\mathrm{ID}} = \mathrm{ID}$**:**
08 $((\delta_i)_{i\in(1,m,\ldots,r)}, \kappa) \leftarrow U_{\mathrm{pend}}$
09 \\ **Update nodes in** $\mathcal{T}_{\mathrm{next}}$
10 $(v_1^n, \ldots, v_r^n) \leftarrow \mathrm{path}(v_{\mathrm{ID}}^n)$
11 $v_i^n.\mathrm{pk} \leftarrow \mathrm{skuPKE.UpdP}(v_i^n.\mathrm{pk}, \Delta_i)$
12 $v_i^n.\mathrm{sk} \leftarrow \mathrm{skuPKE.UpdS}(v_i^n.\mathrm{sk}, \delta_i)$
13 $I_{\mathrm{next}} \leftarrow I_{\mathrm{next}} \oplus \kappa$
14 $U_{\mathrm{pend}} \leftarrow \emptyset$
15 **Else:**
16 $j \leftarrow \min\{i \,:\, v_i^n \in \mathrm{path}(v_{\mathrm{ID}}^n) \cap \mathrm{path}(v_{\tilde{\mathrm{ID}}}^n)\}$
17 **For** $i \in (m, \ldots, j-1)$**:**
18 **If** $\mathrm{ID} \in v_i^n.unm_1$**:**
19 $\kappa \leftarrow \mathsf{ctxt\text{-}decrypt}(\tilde{C}_i, \mathcal{T})$
20 **If** $v_i^n = v_{root}^n$**:**
21 $I_{\mathrm{next}} \leftarrow \kappa$
22 **Else:** $v_i^n.sk \leftarrow \kappa$
23 $\gamma \leftarrow \text{proc-path-upd}(\gamma, (\Delta_i, C_i)_{i\in(1,j,\ldots,r)})$
24 **For** $A_i \in A$**:**
25 $(\text{`}\mathtt{add}(\mathrm{ID}')', {}^{p}\mathcal{T}, (\mathrm{pk}_c, \mathrm{sk}_c), U, e_{\mathrm{ctr}}^{\sim}, \sigma, \tilde{\mathrm{ID}}) \leftarrow A_i$
26 **Require** $\mathsf{Verify}(\mathrm{svk}_{\tilde{\mathrm{ID}}}, \sigma) = 1 \wedge e_{\mathrm{ctr}}^{\sim}[0] = e_{\mathrm{ctr}}[0]$
27 **Execute lines 05 to 23 with input** U
28 $O_{\mathrm{next}} \xleftarrow{\cup} \{\text{`}\mathtt{add}(\mathrm{ID}')'\}$
29 **For** $R_i \in R$**:**
30 $(\text{`}\mathtt{remove}(\mathrm{ID}')', e_{\mathrm{ctr}}, \sigma, \tilde{\mathrm{ID}}) \leftarrow R_i$
31 **Require** $\mathsf{Verify}(\mathrm{svk}_{\tilde{\mathrm{ID}}}, \sigma) = 1 \wedge e_{\mathrm{ctr}}^{\sim}[0] = e_{\mathrm{ctr}}[0]$
32 $O_{\mathrm{next}} \xleftarrow{\cup} \{\text{`}\mathtt{remove}(\mathrm{ID}')'\}$
33 **If** $e_{\mathrm{ctr}} = (e_1, e_2) = (e_1, k-1)$ **:**
34 $e_{\mathrm{ctr}} \leftarrow (e_1 + 1, 0)$
35 $\mathcal{T}_{\mathrm{next}} \leftarrow \text{upd-tree}(\mathcal{T}_{\mathrm{next}}, O_{\mathrm{next}})$
36 $O_{\mathrm{next}} \leftarrow \emptyset$
37 $T \leftarrow \mathcal{T}_{\mathrm{next}}$
38 $K \leftarrow \mathsf{H}_1(\text{`}\mathtt{key}', I_{\mathrm{next}})$
39 $I_{\mathrm{next}} \leftarrow \mathsf{H}_1(\text{`}\mathtt{next}', I_{\mathrm{next}})$
40 **Else** $e_{\mathrm{ctr}} \leftarrow (e_1, e_2 + 1)$
41 $\gamma \leftarrow (\mathrm{ID}, \mathcal{T}, \mathcal{T}_{\mathrm{next}}, O_{\mathrm{next}}, U_{\mathrm{pend}}, e_{\mathrm{ctr}}, (\mathrm{pk}_c, \mathrm{sk}_c), K, I_{\mathrm{next}})$
42 \\ **Case:** ID **is not part of the group yet**
43 **Else:**
44 $(\mathrm{ID}, \mathrm{sk}, \mathrm{ssk}) \leftarrow \gamma$
45 \\ **Sub-case:** ID **is added during the initialization of the group**
46 **If** $W \neq \perp$**:**
47 $(\mathcal{T}, (\Delta_i, C_i)_{i\in(1,\ldots,r)}, (\mathrm{pk}_c, \mathrm{sk}_c), \sigma, \mathrm{ID}_1) \leftarrow W$
48 **Require** $\mathsf{Verify}(\mathrm{svk}_{\mathrm{ID}_1}, \sigma) = 1$
49 $(v_{\mathrm{ID}}.\mathrm{sk}, v_{\mathrm{ID}}.\mathrm{ssk}) \leftarrow (\mathrm{sk}, \mathrm{ssk})$
50 $\gamma \leftarrow (\mathrm{ID}, \mathcal{T}, \mathcal{T}, \emptyset, \emptyset, (0,0), (\mathrm{pk}_c, \mathrm{sk}_c), 0, 0)$
51 $\gamma \leftarrow \text{proc-path-upd}(\gamma, \mathrm{ID}_1, (\Delta_i, C_i)_{i\in(1,\ldots,r)})$
52 $K \leftarrow \mathsf{H}_1(\text{`}\mathtt{key}', I_{\mathrm{next}})$
53 $I_{\mathrm{next}} \leftarrow \mathsf{H}_1(\text{`}\mathtt{next}', I_{\mathrm{next}})$
54 $\gamma \leftarrow (\mathrm{ID}, \mathcal{T}, \mathcal{T}_{\mathrm{next}}, O_{\mathrm{next}}, U_{\mathrm{pend}}, e_{\mathrm{ctr}}, (\mathrm{pk}_c, \mathrm{sk}_c), K, I_{\mathrm{next}})$
55 \\ **Sub-case** ID **is added as part of an add operation**
56 \\ **Let** $B_1^p, \ldots, B_k^p$ **be the blocks from the previous epoch** e
57 **Else:**
58 **Require** $\exists j \in [k]\ ,\ \tilde{A}_\ell \in A \in B_j^p \,:\, \tilde{A}_\ell[0] = \text{`}\mathtt{add}(\mathrm{ID})'$
59 $(\text{`}\mathtt{add}(\mathrm{ID})', {}^{p}\mathcal{T}, (\mathrm{pk}_c, \mathrm{sk}_c), U, e_{\mathrm{ctr}}^{\sim}, \sigma, \tilde{\mathrm{ID}}) \leftarrow \tilde{A}_\ell$
60 **Require** $\mathsf{Verify}(\mathrm{svk}_{\tilde{\mathrm{ID}}}, \sigma) = 1 \wedge e_{\mathrm{ctr}}^{\sim}[0] = e$
61 **Process public part blocks** $B_1^p, \ldots, B_k^p$
(i.e., as in 33 to 35 of proc-path-upd**,**
and 24 to 41 of DeCAF.Proc**)**
62 $(v_{\mathrm{ID}}.\mathrm{sk}, v_{\mathrm{ID}}.\mathrm{ssk}) \leftarrow (\mathrm{sk}, \mathrm{ssk})$
63 $\gamma \leftarrow (\mathrm{ID}, \mathcal{T}, \mathcal{T}, \emptyset, \emptyset, e, (\mathrm{pk}_c, \mathrm{sk}_c), 0, 0)$
64 $\mathrm{DeCAF.Proc}(\gamma, B)$
65 **Return** γ

Figure 6.4: DeCAF Algorithm to process a block. We write the internal state of users not yet part of the groups as $\gamma = (\mathrm{ID}, \mathrm{sk}, \mathrm{ssk})$, i.e., containing their identifier, together with the secret decryption and signing keys.

Alg gen-path-upd(γ)

00 $(\mathrm{ID}, \mathcal{T}, \mathcal{T}_{\text{next}}, O_{\text{next}}, U_{\text{pend}}, e_{\text{ctr}}, (\text{pk}_c, \text{sk}_c), K, I_{\text{next}}) \leftarrow \gamma$
01 $(v_1, \ldots, v_r) \leftarrow \text{path}(v_{\text{ID}})$
02 \\ **Determine height at which user is merged into tree**
03 **If** $u = \max\{i \,:\, \text{ID} \in v_{i-1}.unm_0 \cup v_{i-1}.unm_1\} \neq \bot$**:**
04 $\quad m \leftarrow u$
05 **Else:** $m \leftarrow 2$
06 \\ **Generate seeds and update tokens**
07 $s_1 \leftarrow_{\$}$
08 **For** $i \in (1, m, \ldots, r)$**:**
09 $\quad$ **If** $i = m$ **:** $s_i \leftarrow \mathsf{H}_1(s_1)$
10 $\quad$ **Elseif** $i > m$ **:** $s_i \leftarrow \mathsf{H}_1(s_{i-1})$
11 $\quad (\Delta_i, \delta_i) \leftarrow \text{kuPKE.Sam}(\mathsf{H}_2(s_i))$
12 $\quad C_i \leftarrow \emptyset$
13 \\ **Encrypt seeds**
14 $Z_m \leftarrow \text{Res}(\text{lparent}(v_m)) \cup \text{Res}(\text{rparent}(v_m)) \cup v_m.unm_1 \setminus \{v_{\text{ID}}\}$
15 **For** $z \in Z_m$**:**
16 $\quad C_m \xleftarrow{\cup} \text{kuPKE.Enc}(z.\text{pk}, s_m)$
17 **For** $i \in (m+1, \ldots, r)$**:**
18 $\quad$ **If** $\text{lparent}(v_i) = v_{i-1} \,:\, w_i \leftarrow \text{rparent}(v_i)$
19 $\quad$ **Else:** $w_i \leftarrow \text{lparent}(v_i)$
20 $\quad Z_i \leftarrow \text{Res}(w_i) \cup v_i.unm_1 \setminus w_i.unm_1$
21 $\quad$ **For** $z \in Z_i$**:**
22 $\quad\quad C_i \xleftarrow{\cup} \text{kuPKE.Enc}(z.\text{pk}, s_i)$
23 $\kappa \leftarrow \mathsf{H}_1(s_r)$
24 **Return** $((\Delta_i, \delta_i, C_i)_{i \in (1,m,\ldots,r)}, \kappa)$

Oracle proc-path-upd($\gamma, \tilde{\text{ID}}, (\Delta_i, C_i)_{i \in (1,\ldots,\tilde{r})}$)

25 $(\mathrm{ID}, \mathcal{T}, \mathcal{T}_{\text{next}}, O_{\text{next}}, U_{\text{pend}}, e_{\text{ctr}}, (\text{pk}_c, \text{sk}_c), K, I_{\text{next}}) \leftarrow \gamma$
26 $(v_1^n, \ldots, v_r^n) \leftarrow \text{path}(v_{\text{ID}}^n)$
27 \\ **Determine height at which sender was merged into tree**
28 **If** $u = \max\{i \,:\, \tilde{\text{ID}} \in v_{i-1}^n.unm_0 \cup v_{i-1}^n.unm_1\} \neq \bot$**:**
29 $\quad m \leftarrow u$
30 **Else:** $m \leftarrow 2$
31 $((\Delta_m, C_m), \ldots, (\Delta_r, C_r)) \leftarrow ((\Delta_2, C_2), \ldots, (\Delta_{\tilde{r}}, C_{\tilde{r}}))$
32 \\ **Update public keys**
33 **For** $i \in (1, m, \ldots, r)$**:**
34 $\quad$ **If** $v_i^n.blank = 1$ **:** $v_i^n.\text{pk} \leftarrow \text{skuPKE.UpdP}(\text{pk}_c, \Delta_i)$
35 $\quad$ **Else:** $v_i^n.\text{pk} \leftarrow \text{skuPKE.UpdP}(v_i^n.\text{pk}, \Delta_i)$
36 \\ **Decrypt seed at intersection of paths and update secret keys**
37 $j \leftarrow \min\{i \,:\, v_i^n \in \text{path}(v_{\text{ID}}^n) \cap \text{path}(v_{\tilde{\text{ID}}}^n) \wedge \text{ID} \notin v_i^n.unm_0\}$
38 $s_j \leftarrow \text{ctxt-decrypt}(C_j, \mathcal{T})$
39 **For** $i \in (j, r)$**:**
40 $\quad$ **If** $i \neq j : s_i \leftarrow \mathsf{H}_1(s_{i-1})$
41 $\quad (\Delta_i, \delta_i) \leftarrow \text{kuPKE.Gen}(\mathsf{H}_2(s_i))$
42 $\quad$ **If** $v_i^n.blank$ **:** $v_i.\text{sk} \leftarrow \text{skuPKE.UpdS}(\text{sk}_c, \delta_i)$
43 $\quad$ **Else :** $v_i^n.\text{sk} \leftarrow \text{skuPKE.UpdS}(v_i.\text{sk}, \delta_i)$
44 \\ **Update group key**
45 $\kappa \leftarrow \mathsf{H}_1(s_r)$
46 $I_{\text{next}} \leftarrow I_{\text{next}} \oplus \kappa$
47 \\ **Keep track of which unmerged-users sets need to be updated**
48 $O_{\text{next}} \xleftarrow{\cup} \{v_m, \ldots, v_r\}$
49 $\gamma \leftarrow (\mathrm{ID}, \mathcal{T}, \mathcal{T}_{\text{next}}, O_{\text{next}}, U_{\text{pend}}, e_{\text{ctr}}, (\text{pk}_c, \text{sk}_c), K, I_{\text{next}})$
50 **Return** γ

Figure 6.5: Helper Functions for DeCAF. The function ctxt_decrypt takes as input a list of ciphertexts C encrypting the seed of a given node to all nodes in its resolution and a ratchet tree $\mathcal{T}$, and outputs the decryption of the ciphertext in C that corresponds to a node whose secret key is included in $\mathcal{T}$.

Add. To add a user, when called by ID, the addition algorithm outputs $\tilde{A} = (A, {}^{p}\mathcal{T}, (\text{pk}_c, \text{sk}_c), U, e_{\text{ctr}}, \sigma, \text{ID})$, containing an add request $A =$ '$\texttt{add}(\text{ID}')$', where ID' is the new user. Further, it contains a copy of the public ratchet tree state, the dummy key pair, an update message U generated as described in the previous paragraph, the epoch counter, a signature σ of the message $(A, {}^{p}\mathcal{T}, (\text{pk}_c, \text{sk}_c), U, e_{\text{ctr}})$ under ssk_{ID}, and the identity ID.

Remove. To remove a user, when called by user ID, the removal algorithm outputs $\tilde{R} = (R, e_{\text{ctr}}, \sigma, \text{ID})$, with $R =$ '$\texttt{remove}(\text{ID}')$' for ID' the removed user, and where σ is a signature of (R, e_{ctr}) under ssk_{ID}.

Processing a Block. To process a block, user ID processes a block $B = (W, U, \tilde{A}, \tilde{R})$ consisting of (a potential) welcome message W, update messages $U = (U_1, \ldots, U_{\ell_u})$, add messages $\tilde{A} = (\tilde{A}_1, \ldots, \tilde{A}_{\ell_a})$, and removal messages $\tilde{R} = (\tilde{R}_1, \ldots, \tilde{R}_{\ell_r})$ as follows. We first describe the case of users already in the group. User ID starts by processing the update messages given by the block as follows. Update message U_ℓ for $\ell \in [\ell_u]$ has the form $((\Delta_i, C_i)_i, (\tilde{C})_i, e_{\mathrm{ctr}}, \sigma, \mathrm{ID})$. First, the user checks whether the signature σ verifies under $\mathrm{svk}'_{\mathrm{ID}}$ and that e_{ctr} matches the value stored in $\mathrm{ID}.\gamma$. If one of the checks fails the update is discarded.

If $\mathrm{ID} = \mathrm{ID}'$, i.e., U_ℓ is an update generated by the processing user, ID retrieves from U_{pend} the corresponding update information $((\Delta_i, \delta_i)_i, \kappa)$ with $i = \{1, m, \ldots, r\}$ for some m, deletes it from U_{pend}, and applies it to their update path $\mathrm{path}(v^{\mathrm{n}}_{\mathrm{ID}}) = (v^{\mathrm{n}}_1, \ldots, v^{\mathrm{n}}_r)$ with respect to $\mathcal{T}_{\mathrm{next}}$ as $v^{\mathrm{n}}_i.\mathrm{pk} \leftarrow \mathrm{skuPKE.UpdP}(v^{\mathrm{n}}_i.\mathrm{pk}, \Delta_i)$ and $v^{\mathrm{n}}_i.\mathrm{sk} \leftarrow \mathrm{skuPKE.UpdS}(v^{\mathrm{n}}_i.\mathrm{sk}, \delta_i)$ (note that this updates all key pairs on ID's update path for which the user has access to the secret key). Then they set $I_{\mathrm{next}} \leftarrow I_{\mathrm{next}} \oplus \kappa$. Else, let $v_{u_1}, \ldots, v_{u_t}$ be the nodes in $\mathrm{path}(v_{\mathrm{ID}}) \cap \mathrm{path}(v_{\mathrm{ID}'})$ such that $\mathrm{ID} \in v_{u_i}.unm_1$ and $u_i \geq m$. Then, $\tilde{C}_{u_i}$ contains an encryption of $v_{u_i}.\mathrm{sk}$ under ID's leaf key $v_{\mathrm{ID}}.\mathrm{pk}$. For $i \in [u_1, \ldots, u_t]$, ID uses the corresponding secret key to recover $v_{u_i}.\mathrm{sk}$ and adds it to the node's state $v_{u_i}.\gamma$ in $\mathcal{T}$ and $\mathcal{T}_{\mathrm{next}}$ unless the state already contains a secret key. Then ID calls $\mathrm{ID}.\gamma \leftarrow \text{proc-path-upd}(\mathrm{ID}.\gamma, (\Delta_i, C_i)_i)$, which updates the keys affected by the update in the working copy $\mathcal{T}_{\mathrm{next}}$ of the ratchet tree (note that the secret keys added in the previous step ensure ID is able to decrypt the ciphertext relevant to them), the working copy of the group key, and the list of merges to be implemented at the end of the epoch.

After processing all update operations, ID processes adds $\tilde{A}$ and subsequently removes $\tilde{R}$. First, they check that the signature included in a message verifies and that the message was generated for the current epoch, discarding it if not. In the case of an add message $\tilde{A}_\ell = (A_\ell, {}^p\mathcal{T}, (\mathrm{pk}_c, \mathrm{sk}_c), U, e_{\mathrm{ctr}}, \sigma, \mathrm{ID})$ the user processes the update message U as described above and appends A_ℓ to O_{next}. For valid remove message $\tilde{R}_\ell = (R_\ell, e_{\mathrm{ctr}}, \sigma, \mathrm{ID})$ the request R_ℓ is added to O_{next}. Finally, if B was the last block of an epoch, i.e., B is the ith block with $i = 0 \bmod k$, then ID prepares the transition to the next epoch. To this end, ID recovers from O_{next} the ordered lists of merges $M = (M_1, \ldots, M_{\ell_m})$, adds $A = (A_1, \ldots, A_{L_a})$, and removes $R = (R_1, \ldots, R_{L_r})$ that were included in the blocks of the current epoch. Then they apply these changes to the working copy of the ratchet tree $\mathcal{T}_{\mathrm{next}} \leftarrow \text{upd-tree}(\mathcal{T}_{\mathrm{next}}, A, R, M)$ to be used in the next epoch, update $T \leftarrow \mathcal{T}_{\mathrm{next}}$, increase the epoch counter to $e_{\mathrm{ctr}} \leftarrow ((e_{\mathrm{ctr}})_1 + 1, 0)$, set O_{next} to the empty list, and update the group key to $I \leftarrow \mathsf{H}_1(\texttt{`key'}, I_{\mathrm{next}})$, and afterwards $I_{\mathrm{next}} \leftarrow \mathsf{H}_1(\texttt{`next'}, I_{\mathrm{next}})$.

Let us now describe the second case, that is, that of users not in the group. We distinguish two further cases according to whether ID (a) was added in an add operation

or (b) in the group initialization (i.e., $W \neq \perp$). In case (a) let $B_1^{\mathrm{p}}, \ldots, B_k^{\mathrm{p}}$ be the blocks of the previous epoch. Then one of these blocks contains an add message $\tilde{A} = (A, {}^{p}\mathcal{T}, (\mathrm{pk}_c, \mathrm{sk}_c), U, e_{\mathrm{ctr}}, \sigma, \mathrm{ID})$ with $A =$ '$\mathtt{add}(\mathrm{ID}')$' being the add request for user ID. The user, after validating the signature and epoch, incorporates ${}^{p}\mathcal{T}, (\mathrm{pk}_c, \mathrm{sk}_c)$ in $\mathrm{ID}.\gamma$. As ${}^{p}\mathcal{T}$ is the ratchet tree of the previous epoch, ID brings it up to date by processing, in order, the blocks $B_1^{\mathrm{p}}, \ldots, B_k^{\mathrm{p}}$. Here, as they do not have access to any secret keys of the tree, they only update the public keys. After this operation, $\mathcal{T}$ and its copy $\mathcal{T}_{\mathrm{next}}$ match the current epoch, and the user adds to their secret state their init decryption key and $\mathrm{ssk}_{\mathrm{ID}}$. They then processes the current block $B = (U, A, R)$ as described above.

Finally, assume that ID was added as part of the group initialization, i.e., case (b) above, with $W = ({}^{p}\mathcal{T}, (\Delta_i, C_i)_i, (\mathrm{pk}_c, \mathrm{sk}_c), \sigma, \mathrm{ID}_1)$. In this case ID checks that the signature σ verifies under $\mathrm{svk}_{\mathrm{ID}_1}$, rejecting it if this is not the case. If ID is the user who issued the initialization message, they recover $((\Delta_i, \delta_i)_i, \kappa)$ from their state, apply the update information to their update path, set $I_{\mathrm{next}} \leftarrow \kappa$, and $I \leftarrow \mathsf{H}_1(\text{'}\mathtt{key}\text{'}, I_{\mathrm{next}})$. If ID did not issue the initialization message, they incorporate $({}^{p}\mathcal{T}, (\mathrm{pk}_c, \mathrm{sk}_c))$ in their state, as well as their init decryption key and $\mathrm{ssk}_{\mathrm{ID}}$, set I_{next} to the zero string, and run $\mathrm{ID}'.\gamma \leftarrow \text{proc-path-upd}(\mathrm{ID}'.\gamma, (\Delta_i, C_i)_i)$ to update $\mathcal{T}_{\mathrm{next}}$. I is set to $\mathsf{H}_1(\text{'}\mathtt{key}\text{'}, I_{\mathrm{next}})$, O_{next} is initialized as empty list, as there are no merge, add, or remove operations yet, and $e_{\mathrm{ctr}} \leftarrow (1, 1)$.

Retrieving the Group Key. To extract the current group key, a user ID fetches I from its state, and deletes this value afterwards.

Sending a Transaction. To send a protocol message, ID simply uses the underlying blockchain protocol to send it as a transaction to the blockchain.

Fetching new Blocks. To fetch the last blocks of operations, ID uses the underlying blockchain protocol to retrieve the blocks added to it since it last did.

6.4 Security

6.4.1 Security model and safe predicate

To analyze the modified protocol, we essentially use the security model from [KPPW+21], which allows the adversary to act partially active and fully adaptive (see Section 2.4). The only differences in the setting of baCGKA are that 1) users are processing concurrent messages, and 2) no messages will ever be rejected. Regarding 2) it is however possible that messages get lost and hence, even if a user generated an update it might not process this update.

Defintion 6.4.1 (Asynchronous baCGKA Security)**.** *The security for baCGKA is modeled using a game between a challenger* C *and an adversary* A*. At the beginning of the game, the adversary queries* `create-group`(G) *and the challenger initialises the group* G *with identities* $(\mathrm{ID}_1, \ldots, \mathrm{ID}_{n'})$*. The adversary* A *can then make a sequence of queries, enumerated below, in any arbitrary order. On a high level,* `add-user` *and* `remove-user` *allow the adversary to control the structure of the group, whereas* `store-on-blockchain` *and* `process` *allow it to control the scheduling of the messages. The query* `update` *simulates the refreshing of a local state. Finally,* `start-corrupt` *and* `end-corrupt` *enable the adversary to corrupt the users for a time period. The entire state and random coins of a corrupted user are leaked to the adversary during this period.*

1. `add-user`$(\mathrm{ID}, \mathrm{ID}')$*: a user* ID *requests to add another user* ID′ *to the group.*
2. `remove-user`$(\mathrm{ID}, \mathrm{ID}')$*: a user* ID *requests to remove another user* ID′ *from the group.*
3. `update`(ID)*: the user* ID *requests to refresh its current local state* γ*.*
4. `store-on-blockchain`$(q_1, \ldots, q_l)$*: for queries* $q_1, \ldots, q_l$*, all of which must be actions of the form* $\mathrm{a}_i \in \{$`create-group`, `add-user`, `remove-user`, `update`$\}$ *by some users* ID_i *(for* $i \in [l]$*), this action stores the outputs of the queries in the next block of the blockchain.*
5. `process`(ℓ', ID)*: for* $(B_1, \ldots, B_\ell) \leftarrow \mathrm{baCGKA.Fetch}(\mathrm{ID}.\gamma)$ *and* $\ell' \in [\ell]$*, this action forwards all blocks* $B_1, \ldots, B_{\ell'}$ *to* ID*, who immediately processes them.*
6. `start-corrupt`(ID)*: from now on the entire internal state and randomness of* ID *is leaked to the adversary, with the exception of* $\mathrm{ssk}_{\mathrm{ID}}$*.*[7]
7. `end-corrupt`(ID)*: ends the leakage of user* ID*'s internal state and randomness to the adversary.*
8. `challenge`(ℓ^*)*:* A *picks a block* B_{ℓ^*}*. Let* I_0 *denote the group key that is established by processing the first* ℓ^* *blocks* $B_1, \ldots, B_{\ell^*}$ *in the blockchain and* I_1 *be a fresh random key; if there is no group key established after block* B_{ℓ^*}*,*[8] *then set* $I_0 = I_1 := \bot$*. The challenger tosses a coin* b *and – if the safe predicate below is satisfied – the key* I_b *is given to the adversary (if the predicate is not satisfied the adversary gets nothing).*

[7] Note, we assume all operations to be done instantly, i.e. parties can only be corrupted before or after they have done some operation.

[8] This could happen if the root of the tree is blanked, e.g. if no update was stored on the blockchain yet.

At the end of the game, the adversary outputs a bit b' and wins if $b' = b$. We call a baCGKA scheme (ε, t, Q)-baCGKA-secure if for any adversary A *making at most Q queries of the form* $\texttt{update}(\cdot)$ *and running in time t it holds*

$$\mathsf{Adv}_{\mathsf{baCGKA}}(\mathsf{A}) := |\mathsf{Pr}[1 \leftarrow \mathsf{A}|b = 0] - \mathsf{Pr}[1 \leftarrow \mathsf{A}|b = 1]| < \varepsilon.$$

We define the safe predicate to rule out all trivial winning strategies, such as challenging a block while some current group member is corrupted.

Defintion 6.4.2 (Critical window, safe user)**.** *Let L be the length of the blockchain, C the number of users* A *corrupts throughout the security game, and $\ell^* \in [L]$. For user* ID*, define $q^-_{\mathrm{ID}} \in [Q]_0$ to be maximal such that the following holds:*

- *There exist $c := \lfloor \log(C) \rfloor + 1$ blocks $B_{\ell^1_{\mathrm{ID}}}, \ldots, B_{\ell^c_{\mathrm{ID}}}$ in distinct epochs within the first ℓ^* blocks in the blockchain such that each contains an update query $\mathrm{a}^i_{\mathrm{ID}} := \texttt{update}(\mathrm{ID})$ ($i \in [c]$) that*

  1. *was generated by* ID *in or after query q^-_{ID},*
  2. *is* successful*, i.e. refers to block $B_{\bar{\ell}^i_{\mathrm{ID}}}$ with $\bar{\ell}^i_{\mathrm{ID}} = \ell^i_{\mathrm{ID}} - (\ell^i_{\mathrm{ID}} \mod k)$.*[9]

  If there do not exist c such blocks then we set $q^-_{\mathrm{ID}} = 0$, the first query.

- *There exists a block $B_{\ell^-_{\mathrm{ID}}}$ with $\ell^-_{\mathrm{ID}} \leq \ell^*$ that contains an update $\mathrm{a}^-_{\mathrm{ID}} := \texttt{update}(\mathrm{ID})$ for user* ID *for which 1) and 2) hold, but the entire epoch does not contain any more successful updates for corrupted users. We call such an update a* single *update.*

Furthermore, let q^+_{ID} be the first query that invalidates ID*'s current keys, i.e., in query q^+_{ID},* ID *processes an initial block $B_{\ell^+_{\mathrm{ID}}}$ of some subsequent epoch*[10] *(i.e. $\ell^+_{\mathrm{ID}}/k = \lfloor \ell^+_{\mathrm{ID}}/k \rfloor > \lfloor \ell^*/k \rfloor$) such that one of the blocks $B_{\ell^*+1}, \ldots, B_{\ell^+_{\mathrm{ID}}}$ contains an update $\mathrm{a}^+_{\mathrm{ID}} := \texttt{update}(\mathrm{ID})$ referring to block $B_{\ell^+_{\mathrm{ID}}-k}$. If* ID *does not process any such query then we set $q^+_{\mathrm{ID}} = Q$, the last query.*
We say that the window $[q^-_{\mathrm{ID}}, q^+_{\mathrm{ID}}]$ is critical *for* ID *with respect to challenge ℓ^*. Moreover, if the user* ID *is* not corrupted *at any time point in the critical window, we say that* ID *is* safe *w.r.t. ℓ^*.*

[9] Recall, by definition of the process operation in our protocol, condition 2) is necessary for the update $\mathrm{a}^i_{\mathrm{ID}}$ in block $B_{\ell^i_{\mathrm{ID}}}$ to be indeed processed by users processing block $B_{\ell^i_{\mathrm{ID}}}$.

[10] Recall, in order to be able to process messages in the current epoch, a user keeps the keys of the first round of the current epoch in its state and will only release these keys once it proceeded to the next epoch.

In Section 6.4.3 we discuss a strenghtening of this definition, that our protocol would also satisfy, but which we omit for now for the sake of simplicity. Similar to [KPPW+21], we define a group key as *safe* if all the users in the group are individually safe, i.e., not corrupted in their critical windows.

Defintion 6.4.3 (Safe predicate)**.** *Let I^* be a group key established by processing the first ℓ^* blocks of the blockchain and let G^* be the set of users which end up in the group after block B_{ℓ^*} was processed. Then the key I^* is considered* safe *if for all users $\mathrm{ID} \in G^*$ we have that ID is safe w.r.t. ℓ^* (as per Definition 6.4.2).*

6.4.2 Security of the protocol

Theorem 6.4.4. *If the secretly key-updatable public key encryption scheme used in DeCAF is $(\varepsilon_{\mathrm{Enc}}, t)$-IND-CPA-secure and the used hash functions are modeled as random oracles, then DeCAF is $(O(\varepsilon_{\mathrm{Enc}} \cdot 2(nQ^2)^2), t, Q)$-baCGKA-secure.*

In order to prove Theorem 6.4.4, we first argue that a *safe* group key is not leaked to the adversary via corruption. We make this formal in the following definition and Lemma 6.4.6. In fact, we define leakage of arbitrary secret information which the adversary could potentially learn through corruption.

Defintion 6.4.5 (Secure keys, update information, and seeds)**.** *For a seed s we say s is* leaked *if it is sampled by a user while this user is corrupted, or it is encrypted to the public key associated to a leaked secret key, or s was derived through $s := \mathsf{H}_1(s^-)$ and s^- is leaked.*
A key I_{next} derived through $I_{\mathrm{next}} := I^-_{\mathrm{next}} \oplus \kappa$ is leaked *if it is contained in a user's state while this user is corrupted, or I^-_{next} and κ are both leaked. If I_{next} was derived through $I_{\mathrm{next}} := \mathsf{H}_1(\text{“}\texttt{next}\text{”}, I^-_{\mathrm{next}})$ then it is* leaked *if it is contained in a user's state while this user is corrupted, or I^-_{next} is leaked. A group key I that was derived through $I \leftarrow \mathsf{H}_1(\text{“}\texttt{key}\text{”}, I_{\mathrm{next}})$ is* leaked *if I is contained in a user's state while this user is corrupted, or I_{next} is leaked.*
Let δ be secret update information that was generated by first sampling a seed s, then computing $s' := \mathsf{H}_1^i(s)$ for some $i \in [\lceil \log(n) \rceil]_0$, and then computing $(\Delta, \delta) \leftarrow \mathrm{kuPKE.Sam}(\mathsf{H}_2(s'))$. The secret update information δ is leaked *if δ is contained in a user's state while this user is corrupted, or s' is leaked.*
The secret key sk_c of the dummy key pair $(\mathrm{pk}_c, \mathrm{sk}_c)$ is always considered leaked. For a user's initial key pair $(\mathrm{pk}, \mathrm{sk})$, sk is leaked if sk was in the user's state while the user was corrupted. Let sk' be a secret key that was generated as $\mathrm{sk}' \leftarrow \mathrm{skuPKE.UpdS}(\mathrm{sk}, \delta)$. The key sk' is leaked *if sk' is contained in a user's state while this user is corrupted, or sk and δ are both leaked.*
A secret key/secret update information/seed is called secure *if it is not leaked. We say*

that a corruption of some user ID *does not leak key* sk*, if leakage of* sk *is independent of that corruption of* ID*.*

Remark. *Note that the above definition only defines security for honestly generated secret keys/secret update information/seeds. This is enough for our purpose, since in our security model the adversary can only act through honest users. Furthermore, the definition might look circular at first sight; however, this is not the case since any seed associated with some node in the tree is only encrypted to keys that are associated with nodes* lower *in the tree.*

Lemma 6.4.6. *Assume there are no collisions among seeds, update information and keys throughout the security experiment. If a group key* I^* *is* safe *as per Definition 6.4.3 then it is secure as per Definition 6.4.5.*

In order to prove Lemma 6.4.6, we rely on the fact that the users who can derive the challenge key I^* are exactly those in G^*, where the set of group members G^* is defined to be the users for which either an $\texttt{add-user}(\cdot, \mathrm{ID})$ operation was included in block $\ell^a \leq \ell^* - (\ell^* \mod k)$, or $\mathrm{ID} \in G$ for the initial group set up by $\texttt{create-group}(G)$ (in which case we let $\ell^a = 0$); and such that no $\texttt{remove-user}(\cdot, \mathrm{ID})$ was included in block ℓ^r, with $\ell^a + k - (\ell^a \mod k) \leq \ell^r \leq \ell^* - k - (\ell^* \mod k)$.

Note that, on the one hand, any operation included in a block and accepted by users must come from a user itself, as the adversary is not allowed to create messages itself. On the other hand, since all users share a common view of the blockchain, they will accept the same operations and have the same view of the group members set.

Lemma 6.4.7. *Assume there are no collisions among seeds, update information and keys throughout the security experiment. Then corruption of users not in* G^* *does not leak* I^**.*

Proof. Assume I^* is leaked. We show that I^* must have been leaked through corruption of some user $\mathrm{ID} \in G^*$. By definition, either a user who had I^* in its state was corrupted or the key I^*_{next} used to derive I^* was leaked. In the first case, since all users share a common view of the blockchain and a user holding I^* must have processed the update in which I^* was generated, clearly this user must be in G^* and hence leakage of I^* is independent of any further corruptions of users outside G^*. Now, consider the second case. Similarly, a user holding I^*_{next} in its state must be in G^*, and the same is true for a user holding I^-_{next} if I^*_{next} was derived as $I^*_{\mathrm{next}} := \mathsf{H}_1(\texttt{`next'}, I^-_{\mathrm{next}})$. Hence we consider the case where I^* is leaked because for some I_{next}, which was derived as $I_{\mathrm{next}} = I'_{\mathrm{next}} \oplus \kappa$, both I'_{next} and κ were leaked.

Let $\mathrm{ID} \notin G^*$ and assume for contradiction that ID during the game learns a seed that was used to derive κ. Clearly, since $\mathrm{ID} \notin G^*$, ID cannot have produced κ itself. Let

$\ell \leq \ell^*$ be the last block index such that $\ell \equiv 0 \mod k$, and let $\ell^- = \ell - k$. We must have that either no add-user$(\cdot, \mathrm{ID})$ operation was included in any block before time ℓ, or that a block $\ell^r \leq \ell^-$ contained a remove-user$(\cdot, \mathrm{ID})$ operation. Now, if there was never an add-user$(\cdot, \mathrm{ID})$ before or at time ℓ (for convenience, here we count time in blocks on the blockchain), no seed was ever encrypted to an initkey of ID at any time before ℓ. Moreover, if ID is added to the group after ℓ, it will not be sent any key or new seed until it belongs to the set $v.unm_1$ for some v on the update path of the user generating κ, meaning that at least one update affecting the v took place after ℓ, thus updating its key at this time. Similarly, if such an operation was included in a block in $[\ell + 1, \ell^*]$ (if such an interval exists), ID will still not receive any encryption by block ℓ^*, and will thus learn no seeds used to derive κ either.

Assume, thus, that ID was removed in block ℓ^r. Since the group key I^* is generated w.r.t. time ℓ, there must have been an entire epoch between $[\ell^r, \ell]$ (the first following the epoch to which ℓ^r belongs to, and where any updates took place), where all new secret update information values were encrypted under keys outside the then blanked path of ID. In particular, ID cannot have learnt a seed that was used to derive κ.

This implies that κ was leaked through corruption of a user in G^* at a time when it did not yet process the update generating I^*. By correctness of the scheme, this user must be able to derive I'_{next}, hence I'_{next} is leaked through the same corruption and, hence, leakage of I^* is independent of any corruption of users outside G^*. □

Proof (of Lemma 6.4.6). By Lemma 6.4.7 leakage of the challenge key I^* is independent of corruption of users outside G^*, hence we only have to consider users $id \in G^*$ in the following. Since the challenge group key I^* is safe, all users $\mathrm{ID} \in G^*$ are safe, i.e. not corrupted during their respective critical windows. This implies for every user $\mathrm{ID} \in G^*$ that 1) ID is not corrupted during the current epoch; 2) either ID was not corrupted before it processed B_{ℓ^*}, or ID successfully updated in at least $c := \lfloor \log(C) \rfloor + 1$ epochs before the current one and after its last corruption (where C denotes the number of corrupted parties), or ID had a successful single update in some previous epoch; and 3) after it processed B_{ℓ^*}, either ID was never corrupted again, or an update for ID gets included into a block after B_{ℓ^*} and ID processed the initial block of the subsequent epoch before it's next corruption started.

We will first argue that due to 3), corruption of safe users after they already processed B_{ℓ^*} does not leak the challenge key I^*. To this aim, note that through successfully updating and processing the initial block of the subsequent epoch, a user completely refreshes its state and, in particular, does not have any of the keys associated with the tree established in block B_{ℓ^*} or with any previous tree state in its state, neither does it have any seeds used to derive such keys in its state. Furthermore, all the seeds used to derive the keys in the tree established in B_{ℓ^*} were encrypted to tree states associated

with blocks *before* block B_{ℓ^*}, and the seed used for the successful update was freshly sampled after processing block B_{ℓ^*} and deleted when processing the initial block of the subsequent epoch. On the other hand, if for some node on the update path the associated seed derived during such a successful update is leaked through another user, then also the key associated to that node in the beginning of the respective epoch is already leaked through that user. In other words, while leakage of some update information could allow an adversary who is given the new key to reverse that update and derive the old key, this old key is already leaked through the same corruption that leaked the update information. This proves that corruption of safe users after they processed B_{ℓ^*} does not leak I^*.

Now, consider a node v in the tree established in block B_{ℓ^*} and assume that every party under v, that was corrupted before it processed B_{ℓ^*}, since corruption ended successfully updated in at least i previous epochs or had a successful single update in some previous epoch, and furthermore every party under v, that was corrupted after it processed B_{ℓ^*}, successfully updated after it processed B_{ℓ^*} and processed the initial block of the subsequent epoch before its next corruption starts. We will show by induction on i that if the secret key, which is associated to v (resp. the challenge key in case v is the root) after block B_{ℓ^*} was processed, is leaked, then at least 2^i of the corrupted parties $\{\mathrm{ID}_1, \ldots, \mathrm{ID}_C\}$ have update paths through v. Since for $i = \lfloor \log(C) \rfloor + 1$ we have that $2^i > C$, it follows that the key associated to node v cannot be leaked. Hence, for $v = v_{root}$ we obtain that I^* is secure.

For the inductive argument, note that for $i = 0$ the statement is true since if the key associated to v is leaked there must be at least $1 = 2^0$ corrupted parties with an update path through v. Now, let $i \geq 1$ and assume that the statement holds for all integers smaller than i. Let l be the epoch in which the last of the corrupted parties with update paths through v updates for the ith time or had a successful single update. During this epoch, key sk_v at node v is replaced with $\mathrm{skuPKE.UpdS}(\ldots \mathrm{skuPKE.UpdS}(\mathrm{skuPKE.UpdS}(\mathrm{sk}_v, \delta_1), \delta_2) \ldots, \delta_J)$, where the rerandomization terms δ_j and s_j stem from the J parties which update node v during epoch l. The group key I, on the other hand, which is associated with the root of the tree, is derived as $\mathsf{H}_1(\text{“}\mathtt{key}\text{”}, I_{\mathrm{next}})$ where I_{next} is replaced with $I_{\mathrm{next}} \oplus \bigoplus_{j \in [J]} \kappa_j$. Note that in order for sk_v (resp. I^* if v is the root of the tree) to be leaked it is necessary that the adversary learns all δ_j (resp. κ_j), which implies that for all $j \in [J]$ the seed used to derive δ_j (resp. κ_j) is leaked, i.e. was either derived from a leaked seed, or encrypted to a leaked key. We consider the three cases that after epoch $l-1$ (a) there are at least two nodes v_1, v_2 in the resolution of the parents of v whose associated keys are leaked, (b) there is exactly one node v' in the resolution of the parents of v whose associated key is leaked and at least one update path in epoch l goes through v', and (c) there is exactly one node v' in the resolution of the parents of v whose associated key is leaked and all of the update paths of epoch l do not go through v'. Note that

one of the cases has to occur since otherwise the key associated to v would be secure after epoch l.

Consider case (a). After epoch $l-1$, by minimality of l, it must hold that either 1) every corrupted party under v_1 and v_2 has updated in at least $i-1$ epochs or had a successful single update, or 2) all but one corrupted party under v_1 and v_2 has updated in at least i epochs or had a successful single update. In case 1), we obtain by the induction hypothesis that at least 2^{i-1} corrupted parties have update paths through v_1 and v_2 respectively. In turn there are at least 2^i corrupted parties under v. In case 2), we have that all corrupted users under v_b for some $b \in \{1,2\}$ have successfully updated in at least i epochs preceding $l-1$ or had a successful single update before epoch $l-1$. Furthermore, the number of corrupted users below v_b is strictly smaller than the number of corrupted parties below v. We denote by l' the epoch in which the last of the corrupted parties with update paths through v_b updates for the ith time or had a successful single update and can now do the same case distinction for epoch l' and node v_b.

In case (b), for every update path of epoch l which goes through v' the seed used to derive the δ_j is encrypted to secure keys. Thus, in order for sk_v to be leaked it is necessary that the seeds used to derive the key associated to node v' were leaked as well. This implies that the key associated to v' is leaked even after epoch l. Thus we can set $l' \leftarrow l$ and make the same case distinction for v'.

Now consider case (c) and let v' be the only node in the resolution of the parents of v that has a leaked associated key. Node v' is not part of the update paths of epoch l. Thus, every corrupted party with update path through v' must have updated in at least i epochs before epoch l or had a successful single update before epoch l, and further by definition of l the number of such parties is strictly smaller than the number of corrupted parties below v. Analogous to above let l' denote the epoch in which the last corrupted party under v' updated for the ith time. We can now make the same case distinction as above.

Summing up, if case (a)1) occurs, then at least 2^i of the corrupted parties $\{\mathrm{ID}_1, \ldots, \mathrm{ID}_C\}$ have update paths through v. If, on the other hand, cases (a)2), (b) or (c) occur, then there exist a parent v' of v and an epoch l' such that all corrupted parties under v' updated at least i times or had a single update, and the last to do so did in epoch l'. Note that repeated application of the case distinction reduces the height of node v' in the tree. Thus if we assume that case (a)1) never occurs, at some point we end up with a leaf node v' such that the associated key is leaked and the user associated with that leaf either was not corrupted or updated at least once since its last corruption; in both cases the associated key would be secure. Thus, at some point case (a)1) has to occur, which implies the desired statement.

□

Lemma 6.4.6 in place, the proof of Theorem 6.4.4 follows the security proof from [KPPW+21]. The main difference here is that we reduce baCGKA security of DeCAF to the IND-CPA security of the underlying secretly key-updatable public-key encryption scheme kuPKE as per Definition 6.2.2 (as opposed to IND-CPA security of a simple public-key encryption scheme as in [KPPW+21]). Looking into the details of our protocol, another difference is that the update information for the group key is derived by hashing a seed associated to the root of the challenge tree, but this update information is never encrypted (as opposed to [KPPW+21], where the seed is directly applied to derive the new group key); this slight modification in our current protocol will allow for quite some simplification of the proof from [KPPW+21].

Repeating the entire rather technical argument of [KPPW+21] would be outside the scope of this work; instead we give a high level overview on the proof of [KPPW+21] and discuss how the proof can be adapted.

Proof sketch (of Theorem 6.4.4). The main idea in [KPPW+21] is the following: If H_1 and H_2 are modeled as random oracles, then all the public-key pairs $(\mathrm{pk}, \mathrm{sk})$ sampled through $\mathrm{kuPKE.Gen}$ as well as the update information (Δ_i, δ_i) have the same distribution as if they were sampled independently (to ensure consistency, the random oracles can be programmed accordingly). Furthermore, by Lemma 6.4.6, the challenge key $I^* := \mathsf{H}_1(\text{“}\mathtt{key}\text{”}, I_{\mathrm{next}})$ is secure, i.e. I^* is not contained in a user's state while the user is corrupted and (the seed) I_{next} is secure.

Now, if the adversary never queries a secure seed to the random oracles H_1 and H_2, then the group key I^* is identically distributed to a uniformly random, independent string. Thus, any adversary that has advantage > 0 in breaking the security of DeCAF must query the oracles H_1 or H_2 on some secure seed; we call this event E.[11] As long as E doesn't happen, every secure seed is information-theoretically hidden unless encrypted to some (secure) key. The idea for our (fully black-box) reduction R now is to embed an IND-CPA challenge (with two uniformly random seeds as messages) for kuPKE and hope that the query that makes E turn true will be the seed that was encrypted in the challenge ciphertext; when E turns true, the reduction stops the experiment. To see why this works, note that by Definition 6.4.5, for every secure key pair $(\mathrm{pk}^*, \mathrm{sk}^*)$ there exist ρ, j^-, j^+ with $-1 \leq j^- < \rho \leq j^+ \leq Q$ such that

- $(\mathrm{pk}^*, \mathrm{sk}^*)$ was derived by ρ times updating either some dummy key pair $(\mathrm{pk}_0, \mathrm{sk}_0)$ or an init key of some user; we write $(\mathrm{pk}_\rho, \mathrm{sk}_\rho) := (\mathrm{pk}^*, \mathrm{sk}^*)$,

[11] In fact, this property of our scheme would allow us to prove security based on a weaker security assumption than IND-CPA security for kuPKE, where given an encryption of a random message the adversary has to compute the message.

- secret keys $(\mathrm{sk}_i)_{i\in[j^-+1,j^+]}$ as well as secret update information δ_{j^-}, δ_{j^+} are secure.

Now, as long as E does not happen, the secret update information δ_{j^-}, δ_{j^+} is identically distributed to freshly sampled, independent update information, hence, the reduction can indeed embed an IND-CPA challenge for kuPKE within the baCGKA security experiment.

To bound the security loss involved by our reduction, note that seeds associated to leaves are information-theoretically hidden unless compromised through corruption, and also the respective other message used in the IND-CPA security experiment is information-theoretically hidden as long as E did not happen[12]. Thus, except with negligible probability, whenever the reduction R correctly guessed ρ^*, j^-, j^+ and embedded the challenge key pair $(\mathrm{pk}_\rho, \mathrm{sk}_\rho)$ of the kuPKE challenge and the two seeds at the right position in the challenge tree, then R succeeds in embedding its challenge and turning the adversary into an adversary against IND-CPA security of the kuPKE scheme. More precisely, before the game starts, R guesses uniformly at random the query q^* in which the seed s^* that makes event E turn true is generated. Furthermore, for the key pk^* to which s^* will be encrypted during the game, R guesses uniformly at random the position v^* in the tree as well as the number of updates ρ^* through which the key pair $(\mathrm{pk}^*, \mathrm{sk}^*)$ was derived, as well as the indices j^-, j^+ for the kuPKE challenge. Thus, R succeeds with probability $1/(2nQ^4)$, and additionally taking into account unmerged leaves, and the probability of a collision between the seeds, we end up with a security loss of roughly $2(nQ^2)^2 + (\log(n)Q)^2/|H_2|$, where $|H_2|$ is the size of the range of H_2. □

6.4.3 A stronger safe predicate

The safe predicate in the section above, or, in particular, the definition of critical window, is written with respect to the users corrupted by A since the beginning of the security game. Here, we will briefly argue that, while we presented it like this for simplicity, in practice one would want to consider a stronger version, that takes into account the users corrupted only from the last time a group key was safe. We will argue that such a strenghthening follows easily, if only at the cost of a more convoluted presentation.

Example: A safe group key not covered by the safe predicate. First, to see why the predicate defined above (Definitions 6.4.2 and 6.4.3) is suboptimal, observe

[12]For simplicity of exposition, we ignore the issue of unmerged leaves here; the general case including unmerged leaves and therefore multiple encryptions of the same seed follows by a hybrid argument, losing another multiplicative factor n in security.

that by defining it in such a fashion, we exclude several situations where a key is safe (but would be marked as unsafe by said predicate). This is because it ignores the possibility of healing at some point throughout the game execution, some time before the challenge query. For instance, consider the game execution where the adversary corrupts every user at some point, but does so by corrupting users two by two, in order from left to right, say. Further, A ends each pair of corruptions before starting the next and, moreover, in between each pair of corruptions, A has the last two corrupted users, concurrently, issue two updates each, thus healing their state. I.e., A first corrupts ID_1 and ID_2, ends the corruption of both of them, makes them issue updates q_1, q_2 respectively, calls `store-on-blockchain`(q_1, q_2), makes both users process this last block, then issue new updates q_1', q_2', and then process the block resulting from `store-on-blockchain`(q_1', q_2'). Done that, then A corrupts ID_3 and ID_4, stops the corruption, and proceeds in the same fashion as before, making these two users update twice, before corrupting ID_5 and ID_6, and so on. In this execution of the game, it is clear that the group key will be secure every time a pair of users execute their pair of concurrent updates. However, from the time the adversary has corrupted 4 or more users, the predicate above will consider any future group key insecure, as $C \geq 4$ corruptions would require either $c \geq 3$ concurrent updates or a single update, from each corrupted user. Since each user only ever updates twice, and those updates are concurrent, the safe predicate will indeed never be satisfied.

A stronger safe predicate. This issue, however, can be solved rather easily by introducing a slightly modified, recursive definition of the safe predicate $\mathrm{safe}(\ell^*)$ associated to block ℓ^* (equivalently, to its corresponding epoch). For this, to ℓ^* we associate $\ell^-(\ell^*) < \ell^*$, the last block before ℓ^* that satisfied $\mathrm{safe}(\ell^-)$, where we set $\ell^-(\ell^*) = 0$ if no such block before ℓ^* exists. Now, safe can be defined as in Section 6.4.1, the only difference being that in Definition 6.4.2 the number of corrupted users $C(\ell^*)$ is defined as the number of users A corrupts between $\ell^-(\ell^*)$ and ℓ^* (instead of the number of all users corrupted up to ℓ^*).

In order to see that the proof would carry over to this new predicate, note that we would only need to ensure that Lemma 6.4.6 still holds. Namely, that if the stronger safe predicate holds for key I^*, then I^* is not leaked. This can indeed be showed through an inductive argument on the sequence of secure epochs. Note that the base case, i.e. $\ell^-(\ell^*) = 0$ corresponds to the already existing predicate and is taken care of by the current proof. For the inductive step, one would need to show that key I^* is secure (as per Definition 6.4.5) given that the group key defined by $\ell^-(\ell^*)$ is secure. This follows from the existing proof together with two observations, which we will briefly argue in the paragraphs below. On the one hand, the fact that the ratchet tree defined by processing blocks up to the safe one $\ell^-(\ell^*)$ exclusively contains keys that have not leaked. On the other, the fact that if a seed set by any update included in any

block after $\ell^-(\ell^*)$ is encrypted under a key pk belonging to a tree associated to some block $\tilde{\ell} < \ell^-(\ell^*)$, then pk also belongs to the tree associated to $\ell^-(\ell^*)$. These two observations ensure that the leakage of any key generated during the period between $\ell^-(\ell^*)$ and ℓ^* can be traced back to a corruption taking place during that same period. This, in turn, allows to use essentially the same proof of Lemma 6.4.6 to argue for the inductive step.

To see why the first observation is true, one can look at the simpler case: if u and v are two nodes in the ratchet tree, with u being the child of v, then it is not possible for the secret key at v to be leaked, while the secret key for u is secure (since, by assumption, the group key at $\ell^-(\ell^*)$ is secure, the statement follows). Indeed, let sk_v be leaked and q_v be the time at which A first learnt the value of a secret key at v (and such that from q_v to the present there was no time when A did not have knowledge of the secret key at v). At this time, A must have learnt this key through a corruption, and so must have also learnt the secret key at u at the time. However, since A has knowledge of the key at v throughout the interval from q_v to the time sk_v was set, they, in particular, must also have learnt all seeds used to derive secret update informations updating the key at v during that time. Consider now the different secret update informations evolving the key at u. Any such δ that comes from an update by a user below v is derived from a seed, itself derived by a hash evaluation of a seed that A learnt. For the other δ coming from the other sub-tree under u, the corresponding seed gets encrypted to a key at v, which A also knows, by assumption. This shows that A would also know the key at u, i.e. it is leaked.

The second observation follows easily from the consistency properties that the blockchain ensures, in particular the agreement of all users on the transcript of the execution so far. Indeed, for the statement of the above observation to not be true, an update consistent with the transcript so far up to some block $\hat{\ell} \leq \tilde{\ell}$ would have needed to be included and processed by users in some block between $\tilde{\ell}$ and $\ell^-(\ell^*)$, which is not possible.

CHAPTER 7

Conclusion

In this thesis we explored the problem of better understanding and improving different aspects of continuous group key agreement. We started by seeing that, while blanking is the primarily used method for removing users in CGKA protocols, tainting is a promising alternative which should be further studied (Chapter 3). Better appreciating how both approaches compare under real-life circumstances, both in the non-concurrent and, specially, within the P&C framework (since the latter has not yet been attempted) is an interesting open problem.

We continued by studying the problem of multiple (overlapping) groups (Chapter 4), where we made considerable progress understanding both lower and upper bounds on the communication cost that one could expect from protocols in this setting. Nevertheless, many open problems remain, from formalizing security of such protocols, including e.g. notions of membership privacy, to more thoroughly understanding the impact of known techniques for performing dynamic operations, and whether better ones exist here.

We finished by presenting two solutions for the problem of concurrency in CGKA protocols (Chapters 5 and 6). Importantly, we introduced a new notion of post-compromise security and showed it can bring compelling trade-offs to the CGKA design space. Adapting techniques presented in these protocols to MLS or other CGKA protocols is an interesting open problem. On the one hand, adapting the update-merging feature to MLS could, for example, not only provide a remedy to the inefficiencies introduced by update proposals, but also potentially allow for concurrent commits. On the other, generalizing the partial states technique from CoCoA and the accompanying download cost reduction, to a wider family of CGKA protocols would be a great contribution.

Overall, CGKA and, more widely, secure group messaging, is an exciting research area

with much development well beyond the topics we focused on: PCS and the efficiency of group operations. This is particularly true given the publication of MLS, the first standard for SGM, which is bound to expand the applications for these algorithms and usher in an array of new open problems.

Bibliography

[AAB+21a] Joël Alwen, Benedikt Auerbach, Mirza Ahad Baig, Miguel Cueto, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. Grafting key trees: Efficient key management for overlapping groups. Cryptology ePrint Archive, Report 2021/1158, 2021. https://eprint.iacr.org/2021/1158.

[AAB+21b] Joël Alwen, Benedikt Auerbach, Mirza Ahad Baig, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. Grafting key trees: Efficient key management for overlapping groups. In Kobbi Nissim and Brent Waters, editors, *TCC 2021: 19th Theory of Cryptography Conference, Part III*, volume 13044 of *Lecture Notes in Computer Science*, pages 222–253, Raleigh, NC, USA, November 8–11, 2021. Springer, Heidelberg, Germany.

[AAB+24a] Harold Abelson, Ross Anderson, Steven M Bellovin, Josh Benaloh, Matt Blaze, Jon Callas, Whitfield Diffie, Susan Landau, Peter G Neumann, Ronald L Rivest, Jeffrey I Schiller, Bruce Schneier, Vanessa Teague, and Carmela Troncoso. Bugs in our pockets: the risks of client-side scanning. *Journal of Cybersecurity*, 10(1):tyad020, 01 2024.

[AAB+24b] Michael Anastos, Benedikt Auerbach, Mirza Ahad Baig, Miguel Cueto Noval, Matthew Kwan, Guillermo Pascual-Perez, and Krzysztof Pietrzak. The cost of maintaining keys in dynamic groups with applications to multicast encryption and group messaging. Cryptology ePrint Archive, Paper 2024/1097, 2024. https://eprint.iacr.org/2024/1097.

[AAN+22a] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, and Krzysztof Pietrzak. DeCAF: Decentralizable continuous group key agreement with fast healing. Cryptology ePrint Archive, Report 2022/559, 2022. https://eprint.iacr.org/2022/559.

[AAN+22b] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. CoCoA: Concurrent continuous group key agreement. In Orr Dunkelman and Stefan Dziembowski, editors, *Advances in Cryptology – EUROCRYPT 2022, Part II*, volume 13276 of *Lecture Notes in Computer Science*, pages 815–844, Trondheim, Norway, May 30 – June 3, 2022. Springer, Heidelberg, Germany.

[AAN+22c] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. CoCoA: Concurrent continuous group key agreement. Cryptology ePrint Archive, Report 2022/251, 2022. https://eprint.iacr.org/2022/251.

[ABJM21a] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Collective information security in large-scale urban protests: the case of hong kong. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021: 30th USENIX Security Symposium*, pages 3363–3380. USENIX Association, August 11–13, 2021.

[ABJM21b] Martin R. Albrecht, Jorge Blasco, Rikke Bjerg Jensen, and Lenka Mareková. Mesh messaging in large-scale protests: Breaking Bridgefy. In Kenneth G. Paterson, editor, *Topics in Cryptology – CT-RSA 2021*, volume 12704 of *Lecture Notes in Computer Science*, pages 375–398, Virtual Event, May 17–20, 2021. Springer, Heidelberg, Germany.

[ABR01] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In David Naccache, editor, *Topics in Cryptology – CT-RSA 2001*, volume 2020 of *Lecture Notes in Computer Science*, pages 143–158, San Francisco, CA, USA, April 8–12, 2001. Springer, Heidelberg, Germany.

[ACC+19] Joël Alwen, Margarita Capretto, Miguel Cueto, Chethan Kamath, Karen Klein, Ilia Markov, Guillermo Pascual-Perez, Krzysztof Pietrzak, Michael Walter, and Michelle Yeo. Keep the dirt: Tainted TreeKEM, adaptively and actively secure continuous group key agreement. Cryptology ePrint Archive, Report 2019/1489, 2019. https://eprint.iacr.org/2019/1489.

[ACD19] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology –*

EUROCRYPT 2019, Part I, volume 11476 of *Lecture Notes in Computer Science*, pages 129–158, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany.

[ACDT20] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele Micciancio and Thomas Ristenpart, editors, *Advances in Cryptology – CRYPTO 2020, Part I*, volume 12170 of *Lecture Notes in Computer Science*, pages 248–277, Santa Barbara, CA, USA, August 17–21, 2020. Springer, Heidelberg, Germany.

[ACDT21] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of MLS. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 1463–1483, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[ACH+24] Joël Alwen, Matthew Campagna, Dominik Hartmann, Shuichi Katsumata, Eike Kiltz, Jake Massimo, Marta Mularczyk, Guillermo Pascual-Perez, Thomas Prest, and Peter Schwabe. How Multi-Recipient KEMs can help the Deployment of Post-Quantum Cryptography. *5th NIST PQC Standardization Conference*, 2024.

[ACJM20] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020: 18th Theory of Cryptography Conference, Part II*, volume 12551 of *Lecture Notes in Computer Science*, pages 261–290, Durham, NC, USA, November 16–19, 2020. Springer, Heidelberg, Germany.

[ACK+21] Benedikt Auerbach, Suvradip Chakraborty, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, Michael Walter, and Michelle Yeo. Inverse-sybil attacks in automated contact tracing. In Kenneth G. Paterson, editor, *Topics in Cryptology – CT-RSA 2021*, volume 12704 of *Lecture Notes in Computer Science*, pages 399–421, Virtual Event, May 17–20, 2021. Springer, Heidelberg, Germany.

[ACNPPP23] Benedikt Auerbach, Miguel Cueto Noval, Guillermo Pascual-Perez, and Krzysztof Pietrzak. On the cost of post-compromise security in concurrent continuous group-key agreement. In Guy Rothblum and Hoeteck Wee, editors, *Theory of Cryptography*, pages 271–300, Cham, 2023. Springer Nature Switzerland.

[AEP22] Martin R. Albrecht, Raphael Eikenberg, and Kenneth G. Paterson. Breaking bridgefy, again: Adopting libsignal is not enough. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022: 31st USENIX Security Symposium*, pages 269–286, Boston, MA, USA, August 10–12, 2022. USENIX Association.

[AFM24] Joël Alwen, Georg Fuchsbauer, and Marta Mularczyk. Updatable public-key encryption, revisited. In Marc Joye and Gregor Leander, editors, *Advances in Cryptology – EUROCRYPT 2024, Part VII*, volume 14657 of *Lecture Notes in Computer Science*, pages 346–376, Zurich, Switzerland, May 26–30, 2024. Springer, Heidelberg, Germany.

[AGRS15] Miriyam Aouragh, Seda Gurses, Jara Rocha, and Femke Snelting. FCJ-196 Let's First Get Things Done! On Division of Labour and Techno-political Practices of Delegation in Times of Crisis. *The Fibreculture Journal*, pages 209–238, 12 2015.

[AHK+23] Joël Alwen, Dominik Hartmann, Eike Kiltz, Marta Mularczyk, and Peter Schwabe. Post-quantum multi-recipient public key encryption. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023: 30th Conference on Computer and Communications Security*, pages 1108–1122, Copenhagen, Denmark, November 26–30, 2023. ACM Press.

[AHKM22] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. Server-aided continuous group key agreement. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 69–82, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.

[AHPP23] Benedikt Auerbach, Charlotte Hoffmann, and Guillermo Pascual-Perez. Generic-group lower bounds via reductions between geometric-search problems: With and without preprocessing. In Guy Rothblum and Hoeteck Wee, editors, *Theory of Cryptography*, pages 301–330, Cham, 2023. Springer Nature Switzerland.

[AJM22] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. In Yevgeniy Dodis and Thomas Shrimpton, editors, *Advances in Cryptology – CRYPTO 2022, Part II*, volume 13508 of *Lecture Notes in Computer Science*, pages 34–68, Santa Barbara, CA, USA, August 15–18, 2022. Springer, Heidelberg, Germany.

[Ala08] Shahidul Alam. Majority world: Challenging the west's rhetoric of democracy. *Amerasia Journal*, 34:87–98, 01 2008.

[AMM00] Y.S. Abu-Mostafa and R.J. McEliece. Maximal codeword lengths in Huffman codes. *Computers & Mathematics with Applications*, 39(11):129 – 134, 2000.

[AMPS22] Martin R. Albrecht, Lenka Mareková, Kenneth G. Paterson, and Igors Stepanovs. Four attacks and a proof for Telegram. In *2022 IEEE Symposium on Security and Privacy*, pages 87–106, San Francisco, CA, USA, May 22–26, 2022. IEEE Computer Society Press.

[AMT23] Joël Alwen, Marta Mularczyk, and Yiannis Tselekounis. Fork-resilient continuous group key agreement. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023, Part IV*, volume 14084 of *Lecture Notes in Computer Science*, pages 396–429, Santa Barbara, CA, USA, August 20–24, 2023. Springer, Heidelberg, Germany.

[And22] Patrick D. Anderson. *Cypherpunk Ethics: Radical Ethics for the Digital Age*. Routledge, 2022.

[ANPPP23] Benedikt Auerbach, Miguel Cueto Noval, Guillermo Pascual-Perez, and Krzysztof Pietrzak. On the cost of post-compromise security in concurrent continuous group-key agreement. In Guy N. Rothblum and Hoeteck Wee, editors, *TCC 2023: 21st Theory of Cryptography Conference, Part III*, volume 14371 of *Lecture Notes in Computer Science*, pages 271–300, Taipei, Taiwan, November 29 – December 2, 2023. Springer, Heidelberg, Germany.

[App21] Apple. CSAM Detection - Technical Summary. `https://www.apple.com/child-safety/pdf/CSAM_Detection_Technical_Summary.pdf`, August 2021. Accessed: 17-06-2024.

[Aru19] Chinmayi Arun. On whatsapp, rumours, and lynchings. *Economic and Political Weekly*, 54:30–35, 01 2019.

[ASKP+17] Ruba Abu-Salma, Kat Krol, Simon Parkin, Victoria Koh, Kevin Kwan, Jazib Mahboob, Zahra Traboulsi, and Angela Sasse. The security blanket of the chat world: An analytic evaluation and a user study of telegram. In *European Workshop on Usable Security EuroUSEC'17*, 04 2017.

[ASSB+17] Ruba Abu-Salma, M. Angela Sasse, Joseph Bonneau, Anastasia Danilova, Alena Naiakshina, and Matthew Smith. Obstacles to the adoption of secure communication tools. In *2017 IEEE Symposium*

on Security and Privacy, SP 2017 - Proceedings, Proceedings - IEEE Symposium on Security and Privacy, pages 137–153. Institute of Electrical and Electronics Engineers Inc., June 2017. Publisher Copyright: © 2017 IEEE.; 2017 IEEE Symposium on Security and Privacy, SP 2017 ; Conference date: 22-05-2017 Through 24-05-2017.

[AW23] Kyoichi Asano and Yohei Watanabe. Updatable public key encryption with strong CCA security: Security analysis and efficient generic construction. Cryptology ePrint Archive, Paper 2023/976, 2023. https://eprint.iacr.org/2023/976.

[BA23] Jenny Blessing and Ross Anderson. One Protocol to Rule Them All? On Securing Interoperable Messaging. In *Security Protocols XXVIII: 28th International Workshop, Cambridge, UK, March 27–28, 2023, Revised Selected Papers*, page 174–192, Berlin, Heidelberg, 2023. Springer-Verlag.

[BBN19] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS. Research report, Inria Paris, December 2019.

[BBR18] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups. https://mailarchive.ietf.org/arch/attach/mls/pdf1XUH6o.pdf, May 2018.

[BBR+23] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. RFC 9420, July 2023.

[BCG23] David Balbás, Daniel Collins, and Phillip Gajland. WhatsUpp with sender keys? Analysis, improvements and security proofs. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology – ASIACRYPT 2023, Part V*, volume 14442 of *Lecture Notes in Computer Science*, pages 307–341, Guangzhou, China, December 4–8, 2023. Springer, Heidelberg, Germany.

[BCK21] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Cryptographic security of the mls rfc, draft 11. Cryptology ePrint Archive, Report 2021/137, 2021. https://eprint.iacr.org/2021/137.

[BCP02] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. Dynamic group Diffie-Hellman key exchange under standard assumptions. In

Lars R. Knudsen, editor, *Advances in Cryptology – EUROCRYPT 2002*, volume 2332 of *Lecture Notes in Computer Science*, pages 321–336, Amsterdam, The Netherlands, April 28 – May 2, 2002. Springer, Heidelberg, Germany.

[BCV23] David Balbás, Daniel Collins, and Serge Vaudenay. Cryptographic administration for secure group messaging. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1253–1270, Anaheim, CA, August 2023. USENIX Association.

[BD95] Mike Burmester and Yvo Desmedt. A secure and efficient conference key distribution system (extended abstract). In Alfredo De Santis, editor, *Advances in Cryptology – EUROCRYPT'94*, volume 950 of *Lecture Notes in Computer Science*, pages 275–286, Perugia, Italy, May 9–12, 1995. Springer, Heidelberg, Germany.

[BDG+22] Alexander Bienstock, Yevgeniy Dodis, Sanjam Garg, Garrison Grogan, Mohammad Hajiabadi, and Paul Rösler. On the worst-case inefficiency of CGKA. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022: 20th Theory of Cryptography Conference, Part II*, volume 13748 of *Lecture Notes in Computer Science*, pages 213–243, Chicago, IL, USA, November 7–10, 2022. Springer, Heidelberg, Germany.

[BDR20] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020: 18th Theory of Cryptography Conference, Part II*, volume 12551 of *Lecture Notes in Computer Science*, pages 198–228, Durham, NC, USA, November 16–19, 2020. Springer, Heidelberg, Germany.

[BDT22] Alexander Bienstock, Yevgeniy Dodis, and Yi Tang. Multicast key agreement, revisited. In Steven D. Galbraith, editor, *Topics in Cryptology – CT-RSA 2022*, volume 13161 of *Lecture Notes in Computer Science*, pages 1–25, Virtual Event, March 1–2, 2022. Springer, Heidelberg, Germany.

[Bel] Luca Belli. WhatsApp skewed Brazilian election, showing social media's danger to democracy. *The Conversation*. `https://theconversation.com/whatsapp-skewed-brazilian-election-showing-social-medias-danger-to-democracy-106476`.

[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, WPES '04, page 77–84, New York, NY, USA, 2004. Association for Computing Machinery.

[BGD+24] Divyanshu Bhardwaj, Carolyn Guthoff, Adrian Dabrowski, Sascha Fahl, and Katharina Krombholz. Mental models, expectations and implications of client-side scanning: An interview study with experts. In *Proceedings of the CHI Conference on Human Factors in Computing Systems*, CHI '24, New York, NY, USA, 2024. Association for Computing Machinery.

[Bob07] Chris Bobel. 'I'm not an activist, though I've done a lot of it': Doing Activism, Being Activist and the 'Perfect Standard' in a Contemporary Movement. *Social Movement Studies*, 6:147–159, 09 2007.

[Box76] George E. P. Box. Science and statistics. *Journal of the American Statistical Association*, 71(356):791–799, 1976.

[BRT23] Alexander Bienstock, Paul Rösler, and Yi Tang. ASMesh: Anonymous and secure messaging in mesh networks using stronger, anonymous double ratchet. In Weizhi Meng, Christian Damsgaard Jensen, Cas Cremers, and Engin Kirda, editors, *ACM CCS 2023: 30th Conference on Computer and Communications Security*, pages 1–15, Copenhagen, Denmark, November 26–30, 2023. ACM Press.

[BSJ+17] Mihir Bellare, Asha Camper Singh, Joseph Jaeger, Maya Nyayapati, and Igors Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part III*, volume 10403 of *Lecture Notes in Computer Science*, pages 619–650, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[BSJCU21] Maia J. Boyd, Jamar L. Sullivan Jr., Marshini Chetty, and Blase Ur. Understanding the security and privacy advice given to black lives matter protesters. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*, CHI '21, New York, NY, USA, 2021. Association for Computing Machinery.

[Cab23] Cable.co.uk. The cost of 1GB of mobile data in 237 countries. `https://www.cable.co.uk/mobiles/worldwide-data-pricing/`, 2023. Accessed: 17-06-2024.

[Cas12] Manuel Castells. *Networks of Outrage and Hope: Social Movements in the Internet Age*. John Wiley & Sons, 2012.

[CCG16] Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. On post-compromise security. In Michael Hicks and Boris Köpf, editors, *CSF 2016: IEEE 29th Computer Security Foundations Symposium*, pages 164–178, Lisbon, Portugal, June 27–1, 2016. IEEE Computer Society Press.

[CCG+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 David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018: 25th Conference on Computer and Communications Security*, pages 1802–1819, Toronto, ON, Canada, October 15–19, 2018. ACM Press.

[CEST24] Kelong Cong, Karim Eldefrawy, Nigel P. Smart, and Ben Terner. The key lattice framework for concurrent group messaging. In Christina Pöpper and Lejla Batina, editors, *ACNS 24: 22nd International Conference on Applied Cryptography and Network Security, Part II*, volume 14584 of *Lecture Notes in Computer Science*, pages 133–162, Abu Dhabi, UAE, March 5–8, 2024. Springer, Heidelberg, Germany.

[CFKN20] Cas Cremers, Jaiden Fairoze, Benjamin Kiesl, and Aurora Naska. Clone detection in secure messaging: Improving post-compromise security in practice. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1481–1495, Virtual Event, USA, November 9–13, 2020. ACM Press.

[CGI+99] Ran Canetti, Juan A. Garay, Gene Itkis, Daniele Micciancio, Moni Naor, and Benny Pinkas. Multicast security: A taxonomy and some efficient constructions. In *IEEE INFOCOM'99*, pages 708–716, New York, NY, USA, March 21–25, 1999.

[CHK21] Cas Cremers, Britta Hale, and Konrad Kohbrok. The complexities of healing in secure group messaging: Why cross-group effects matter. In Michael Bailey and Rachel Greenstadt, editors, *USENIX Security 2021: 30th USENIX Security Symposium*, pages 1847–1864. USENIX Association, August 11–13, 2021.

[CLMP24] Céline Chevalier, Guirec Lebrun, Ange Martinelli, and Jérôme Plût. The art of bonsai: How well-shaped trees improve the communication cost of MLS. Cryptology ePrint Archive, Paper 2024/746, 2024. `https://eprint.iacr.org/2024/746`.

[Coi] XX Coin. Elixxir architecture brief v2.0. https://xx.network/elixxir-architecture-brief-v1.0.pdf.

[CPZ20] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The Signal private group system and anonymous credentials supporting efficient verifiable encryption. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020: 27th Conference on Computer and Communications Security*, pages 1445–1459, Virtual Event, USA, November 9–13, 2020. ACM Press.

[DB05] Ratna Dutta and Rana Barua. Dynamic group key agreement in tree-based setting. In Colin Boyd and Juan Manuel González Nieto, editors, *ACISP 05: 10th Australasian Conference on Information Security and Privacy*, volume 3574 of *Lecture Notes in Computer Science*, pages 101–112, Brisbane, Queensland, Australia, July 4–6, 2005. Springer, Heidelberg, Germany.

[DDF21] Julien Devigne, Céline Duguey, and Pierre-Alain Fouque. MLS group messaging: How zero-knowledge can secure updates. In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, *ESORICS 2021: 26th European Symposium on Research in Computer Security, Part II*, volume 12973 of *Lecture Notes in Computer Science*, pages 587–607, Darmstadt, Germany, October 4–8, 2021. Springer, Heidelberg, Germany.

[Di 20] Philip Di Salvo. *Digital Whistleblowing Platforms in Journalism: Encrypting Leaks*. Palgrave Macmillan, 01 2020.

[DKW21] Yevgeniy Dodis, Harish Karthikeyan, and Daniel Wichs. Updatable public key encryption in the standard model. In Kobbi Nissim and Brent Waters, editors, *TCC 2021: 19th Theory of Cryptography Conference, Part III*, volume 13044 of *Lecture Notes in Computer Science*, pages 254–285, Raleigh, NC, USA, November 8–11, 2021. Springer, Heidelberg, Germany.

[DNDS19] Sergej Dechand, Alena Naiakshina, Anastasia Danilova, and Matthew Smith. In Encryption We Don't Trust: The Effect of End-to-End Encryption to the Masses on User Perception. In *2019 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 401–415, 06 2019.

[DV19] F. Betül Durak and Serge Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. In Nuttapong Attrapadung and Takeshi Yagi, editors, *IWSEC 19: 14th International Workshop*

on Security, Advances in Information and Computer Security, volume 11689 of *Lecture Notes in Computer Science*, pages 343–362, Tokyo, Japan, August 28–30, 2019. Springer, Heidelberg, Germany.

[DVS] Alex Davidson, Fernando Virdia, and Luiza Soezima. Securing semi-open group messaging. *CAW 2024*. https://fundamental.domains/presentations/caw24/caw.pdf.

[DY83] D. Dolev and A. Yao. On the security of public key protocols. *IEEE Transactions on Information Theory*, 29(2):198–208, 1983.

[EE] EFF and EDRi. Statement on the future of the CSA Regulation. https://edri.org/wp-content/uploads/2024/07/Statement_-The-future-of-the-CSA-Regulation.pdf. Accessed: 08.07.2024.

[EFF16] EFF. Secure Messaging Scorecard. https://www.eff.org/es/pages/secure-messaging-scorecard, 2016.

[EFF18] EFF. Why We Can't Give You A Recommendation. https://www.eff.org/deeplinks/2018/03/why-we-cant-give-you-recommendation, March 2018.

[EHM17] Ksenia Ermoshina, Harry Halpin, and Francesca Musiani. Can Johnny build a protocol? Co-ordinating developer and user intentions for privacy-enhanced secure messaging protocols. In *European Workshop on Usable Security '17*, 01 2017.

[EJKM22] Edward Eaton, David Jao, Chelsea Komlo, and Youcef Mokrani. Towards post-quantum key-updatable public-key encryption via supersingular isogenies. In Riham AlTawy and Andreas Hülsing, editors, *SAC 2021: 28th Annual International Workshop on Selected Areas in Cryptography*, volume 13203 of *Lecture Notes in Computer Science*, pages 461–482, Virtual Event, September 29 – October 1, 2022. Springer, Heidelberg, Germany.

[EKN+22] Keita Emura, Kaisei Kajita, Ryo Nojima, Kazuto Ogawa, and Go Ohtake. Membership privacy for asynchronous group messaging. Cryptology ePrint Archive, Report 2022/046, 2022. https://eprint.iacr.org/2022/046.

[EM21] Ksenia Ermoshina and Francesca Musiani. The telegram ban: How censorship "made in russia" faces a global internet. *First Monday*, 26(5), Apr. 2021.

[EM22] Ksenia Ermoshina and Francesca Musiani. *Concealing for Freedom: The Making of Encryption, Secure Messaging and Digital Liberties*. Mattering Press, Manchester, 2022.

[EM23] Ksenia Ermoshina and Francesca Musiani. Encryption as a battleground in Ukraine. In Corinne Cath, editor, *Eaten by the Internet*, pages 82–88. Meatspace Press, 2023.

[FK23] Matthias Fassl and Katharina Krombholz. Why I Can't Authenticate — Understanding the Low Adoption of Authentication Ceremonies with Autoethnography. In *Proceedings of the 2023 CHI Conference on Human Factors in Computing Systems*, CHI '23, New York, NY, USA, 2023. Association for Computing Machinery.

[FN21] Sharon Bradford Franklin and Greg Nojeim. International Coalition Calls on Apple to Abandon Plan to Build Surveillance Capabilities into iPhones, iPads, and other Products. `https://cdt.org/insights/international-coalition-calls-on-apple-to-abandon-plan-to-build-surveillance-capabilities-into-iphones-ipads-and-other-products/`, 09 2021. Accessed: 17-06-2024.

[Fou] The Matrix.org Foundation. Matrix specification. `https://matrix.org/docs/spec/`.

[Gab99] Xavier Gabaix. Zipf's law for cities: An explanation. *The Quarterly Journal of Economics*, 114(3):739–7675, 1999.

[GdZAACR21] Homero Gil de Zúñiga, Alberto Ardèvol-Abreu, and Andreu Casero-Ripollés. WhatsApp Political Discussion, Conventional Participation and Activism: Exploring Direct, Indirect and Generational Effects. *Information Communication and Society*, 24:201–218, 01 2021.

[GMS+18] Tamy Guberek, Allison Mcdonald, Sylvia Simioni, Abraham Mhaidli, Kentaro Toyama, and Florian Schaub. Keeping a Low Profile? Technology, Risk and Privacy among Undocumented Immigrants. *ACM SIGCHI Bulletin*, 04 2018.

[GT24] Timnit Gebru and Émile Torres. The TESCREAL bundle: Eugenics and the promise of utopia through artificial general intelligence. *First Monday*, 04 2024.

[HCS20] Alison Harcourt, George Christou, and Seamus Simpson. *Global Standard Setting in Internet Governance*. Oxford University Press, 01 2020.

[HEM18] Harry Halpin, Ksenia Ermoshina, and Francesca Musiani. *Co-ordinating Developers and High-Risk Users of Privacy-Enhanced Secure Messaging Protocols: 4th International Conference, SSR 2018, Darmstadt, Germany, November 26-27, 2018, Proceedings*, pages 56–75. 11 2018.

[HK22] Britta Hale and Chelsea Komlo. On end-to-end encryption. Cryptology ePrint Archive, Report 2022/449, 2022. https://eprint.iacr.org/2022/449.

[HKK23] Dennis Hofheinz, Julia Kastner, and Karen Klein. The power of undirected rewindings for adaptive security. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023, Part II*, volume 14082 of *Lecture Notes in Computer Science*, pages 725–758, Santa Barbara, CA, USA, August 20–24, 2023. Springer, Heidelberg, Germany.

[HKP+21] Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. A concrete treatment of efficient continuous group key agreement via multi-recipient PKEs. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 1441–1462, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[HKP22] Keitaro Hashimoto, Shuichi Katsumata, and Thomas Prest. How to hide MetaData in MLS-like secure group messaging: Simple, modular, and post-quantum. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 1399–1412, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.

[HLA19] Chris Howell, Tom Leavy, and Joël Alwen. Wickr messaging protocol : Technical paper, 2019. https://1c9n2u3hx1x732fbvk1ype2x-wpengine.netdna-ssl.com/wp-content/uploads/2019/12/WhitePaper_WickrMessagingProtocol.pdf.

[HLP22] Calvin Abou Haidar, Benoît Libert, and Alain Passelègue. Updatable public key encryption from DCR: Efficient constructions with stronger security. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022: 29th Conference on Computer and Communications Security*, pages 11–22, Los Angeles, CA, USA, November 7–11, 2022. ACM Press.

[Hor17] Jonathan Horowitz. Who Is This "We" You Speak of? Grounding Activist Identity in Social Psychology. *Socius*, 3:2378023117717819, 2017. PMID: 30221196.

[HPS23] Calvin Abou Haidar, Alain Passelègue, and Damien Stehlé. Efficient updatable public-key encryption from lattices. In Jian Guo and Ron Steinfeld, editors, *Advances in Cryptology – ASIACRYPT 2023, Part V*, volume 14442 of *Lecture Notes in Computer Science*, pages 342–373, Guangzhou, China, December 4–8, 2023. Springer, Heidelberg, Germany.

[Huf52] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[Hug93] Eric Hughes. A Cypherpunk's Manifesto. 1993.

[HZ15] Gulizar Haciyakupoglu and Weiyu Zhang. Social Media and Trust during the Gezi Protests in Turkey. *Journal of Computer-Mediated Communication*, 20, 03 2015.

[IAV22] Rawane Issa, Nicolas Alhaddad, and Mayank Varia. Hecate: Abuse reporting in secure messengers with sealed sender. In Kevin R. B. Butler and Kurt Thomas, editors, *USENIX Security 2022: 31st USENIX Security Symposium*, pages 2335–2352, Boston, MA, USA, August 10–12, 2022. USENIX Association.

[IRC15] Iulia Ion, Rob Reeder, and Sunny Consolvo. "...No one can hack my mind": comparing expert and non-expert security practices. In *Proceedings of the Eleventh USENIX Conference on Usable Privacy and Security*, SOUPS '15, page 327–346, USA, 2015. USENIX Association.

[107] International Telecommunication Union (ITU). The World in 2010: The rise of 3G. `https://www.itu.int/ITU-D/ict/material/FactsFigures2010.pdf`, 2010.

[ITW82] I. Ingemarsson, D. Tang, and C. Wong. A conference key distribution system. *IEEE Transactions on Information Theory*, 28(5):714–720, 1982.

[JCKT20] Rikke Bjerg Jensen, Lizzie Coles-Kemp, and Reem Talhouk. When the Civic Turn turns Digital: Designing Safe and Secure Refugee Resettlement. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, page 1–14, New York, NY, USA, 2020. Association for Computing Machinery.

[JKK+17] Zahra Jafargholi, Chethan Kamath, Karen Klein, Ilan Komargodski, Krzysztof Pietrzak, and Daniel Wichs. Be adaptive, avoid overcommitting. In Jonathan Katz and Hovav Shacham, editors, *Advances in Cryptology – CRYPTO 2017, Part I*, volume 10401 of *Lecture Notes in Computer Science*, pages 133–163, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[JMM19] Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Yuval Ishai and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2019, Part I*, volume 11476 of *Lecture Notes in Computer Science*, pages 159–188, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany.

[JS18] Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 33–62, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[Juk12] Stasys Jukna. *Boolean function complexity: advances and frontiers*, volume 27. Springer Science & Business Media, 2012.

[Kaz15] Becky Kazansky. FCJ-195 Privacy, Responsibility, and Human Rights Activism. *The Fibreculture Journal*, pages 190–208, 12 2015.

[Kel98] Thomas Kelly. The myth of the skytale. *Cryptologia*, 22(3):244–260, 1998.

[KKPP20] Shuichi Katsumata, Kris Kwiatkowski, Federico Pintore, and Thomas Prest. Scalable ciphertext compression techniques for post-quantum KEMs and their applications. In Shiho Moriai and Huaxiong Wang, editors, *Advances in Cryptology – ASIACRYPT 2020, Part I*, volume 12491 of *Lecture Notes in Computer Science*, pages 289–320, Daejeon, South Korea, December 7–11, 2020. Springer, Heidelberg, Germany.

[KL14] Jonathan Katz and Yehuda Lindell. *Introduction to Modern Cryptography*. CRC Press, second edition, 2014.

[Kle21] Karen Klein. *On the adaptive security of graph-based games*. PhD thesis, Institute of Science and Technology Austria, 2021. 10.15479/at:ista:10035.

[KNC20] Yong Ming Kow, Bonnie Nardi, and Wai Kuen Cheng. Be water: Technologies in the leaderless anti-elab movement in hong kong. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, page 1–12, New York, NY, USA, 2020. Association for Computing Machinery.

[Kno23] Mallory Knodel. Encryption regulation, and what to do about it? In Corinne Cath, editor, *Eaten by the Internet*, pages 89–95. Meatspace Press, 2023.

[Kno24] Mallory Knodel. Children's Rights at the Centre of Digital Technology Standards by Design. `https://www.orfonline.org/expert-speak/children-s-rights-at-the-centre-of-digital-technology-standards-by-design`, 02 2024. Accessed: 17-06-2024.

[Kob18] Nadim Kobeissi. *Formal verification for real-world cryptographic protocols and implementations*. Phd thesis, Université Paris sciences et lettres, Dec 2018.

[KPPW+21] Karen Klein, Guillermo Pascual-Perez, Michael Walter, Chethan Kamath, Margarita Capretto, Miguel Cueto, Ilia Markov, Michelle Yeo, Joël Alwen, and Krzysztof Pietrzak. Keep the dirt: Tainted TreeKEM, adaptively and actively secure continuous group key agreement. In *2021 IEEE Symposium on Security and Privacy*, pages 268–284, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.

[Kun09] Meglena Kuneva. Keynote Speech - Roundtable on Online Data Collection, Targeting and Profiling. `https://ec.europa.eu/commission/presscorner/detail/en/SPEECH_09_156`, March 2009.

[LA10] Jeroen Laer and Peter Aelst. Internet and social movement action repertoires. *Information, Communication & Society*, pages 1146–1171, 12 2010.

[LC16] Francis L.F. Lee and Joseph Man Chan. Digital media activities and mode of participation in a protest campaign: a study of the Umbrella Movement. *Information, Communication & Society*, 19(1):4–22, 2016.

[LCC20] Francis Lee, Michael Chan, and Hsuan-Ting Chen. Social Media and Protest Attitudes During Movement Abeyance: A Study of Hong Kong University Students. *International Journal of Communication*, 14:4932–4951, 09 2020.

[LCG+24] Julia Len, Melissa Chase, Esha Ghosh, Kim Laine, and Radames Cruz Moreno. OPTIKS: An Optimized Key Transparency System. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, aug 2024.

[LGGR23] Julia Len, Esha Ghosh, Paul Grubbs, and Paul Rösler. Interoperability in end-to-end encrypted messaging. Cryptology ePrint Archive, Paper 2023/386, 2023. https://eprint.iacr.org/2023/386.

[LHK+20] Ada Lerner, Helen Yuxun He, Anna Kawakami, Silvia Catherine Zeamer, and Roberto Hoyle. Privacy and activism in the transgender community. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*, CHI '20, page 1–13, New York, NY, USA, 2020. Association for Computing Machinery.

[Luh16] Alec Luhn. Russia passes 'big brother' anti-terror laws. https://www.theguardian.com/world/2016/jun/26/russia-passes-big-brother-anti-terror-laws, June 2016.

[LZR17] Ada Lerner, Eric Zeng, and Franziska Roesner. Confidante: Usable Encrypted Email: A Case Study with Lawyers and Journalists. In *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*, pages 385–400, 04 2017.

[Mat19] Matthew A. Weidner. Group Messaging for Secure Asynchronous Collaboration. Master's thesis, University of Cambridge, June 2019.

[MCHR15] Susan E. McGregor, Polina Charters, Tobin Holliday, and Franziska Roesner. Investigating the computer security practices and needs of journalists. In *24th USENIX Security Symposium (USENIX Security 15)*, pages 399–414, Washington, D.C., August 2015. USENIX Association.

[McL16] Jenna McLaughlin. Report: Arab gulf states are surveiling, imprisoning, and silencing activists for social media posts. https://web.archive.org/web/20201015114424/https://theintercept.com/2016/11/01/report-arab-gulf-states-are-surveiling-imprisoning-and-silencing-activists-for-social-media-posts/, 2016.

[Met22] Meta. Independent Assessment: Expanding End-to-End Encryption Protects Fundamental Human Rights. https://about.fb.com/news/2022/04/expanding-end-to-end-encryption-

protects-fundamental-human-rights/, 04 2022. Accessed: 17-06-2024.

[MFK16] Susan E. McGregor, Roesner Franziska, and Caine Kelly. Individual versus organizational computer security and privacy concerns in journalism. *Proceedings on Privacy Enhancing Technologies*, 2016(4):418–435, 07 2016.

[MJHY15] Helen Margetts, Peter John, Scott Hale, and Taha Yasseri. *Political Turbulence: How Social Media Shape Collective Action*. Princeton University Press, 11 2015.

[MKKS+23] Harjasleen Malvai, Lefteris Kokoris-Kogias, Alberto Sonnino, Esha Ghosh, Ercan Oztürk, Kevin Lewi, and Sean F. Lawlor. Parakeet: Practical key transparency for end-to-end encrypted messaging. In *30th Annual Network and Distributed System Security Symposium, NDSS 2023, San Diego, California, USA, February 27 - March 3, 2023*. The Internet Society, 2023.

[MNP18] Mette Mortensen, Christina Neumayer, and Thomas Poell. *Social Media Materialities and Protest: Critical Reflections*. Routledge, 2018.

[MP04] Daniele Micciancio and Saurabh Panjwani. Optimal communication complexity of generic multicast key distribution. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 153–170, Interlaken, Switzerland, May 2–6, 2004. Springer, Heidelberg, Germany.

[MP16] Moxie Marlinspike and Trevor Perrin. The Double Ratchet Algorithm. https://signal.org/docs/specifications/doubleratchet/, November 2016.

[MRH04] Ueli M. Maurer, Renato Renner, and Clemens Holenstein. Indifferentiability, impossibility results on reductions, and applications to the random oracle methodology. In Moni Naor, editor, *TCC 2004: 1st Theory of Cryptography Conference*, volume 2951 of *Lecture Notes in Computer Science*, pages 21–39, Cambridge, MA, USA, February 19–21, 2004. Springer, Heidelberg, Germany.

[MSAAA14] Trust Tshepo Mapoka, Simon Shepherd, Raed Abd-Alhameed, and Kelvin OO Anoh. Novel rekeying approach for secure multiple multicast groups over wireless mobile networks. In *2014 International Wireless Communications and Mobile Computing Conference (IWCMC)*, pages 839–844. IEEE, 2014.

[MSR20] Bill Marczak and John Scott-Railton. Move Fast and Roll Your Own Crypto: A Quick Look at the Confidentiality of Zoom Meetings. Technical report, Citizen Lab, 04 2020. Accessed: 17-06-2024.

[Muk20] Rahul Mukherjee. Mobile witnessing on WhatsApp: Vigilante virality and the anatomy of mob lynching. *South Asian Popular Culture*, 18:1–23, 04 2020.

[MV23] Marino Miculan and Nicola Vitacolonna. Automated verification of telegram's mtproto 2.0 in the symbolic model. *Computers & Security*, 126:103072, 2023.

[New19] The Stand News. In hong kong, authorities arrest the administrator of a telegram protest group and force him to hand over a list of its members. https://web.archive.org/web/20240627141400/https://globalvoices.org/2019/06/14/in-hong-kong-authorities-arrest-the-administrator-of-a-telegram-protest-group-and-force-him-to-hand-over-a-list-of-its-members/, 2019.

[OoPA] U.S. Department of Justice Office of Public Affairs. International statement: End-to-end encryption and public safety. =https://www.justice.gov/opa/pr/international-statement-end-end-encryption-and-public-safety.

[Pan07] Saurabh Panjwani. Tackling adaptive corruptions in multicast encryption protocols. In Salil P. Vadhan, editor, *TCC 2007: 4th Theory of Cryptography Conference*, volume 4392 of *Lecture Notes in Computer Science*, pages 21–40, Amsterdam, The Netherlands, February 21–24, 2007. Springer, Heidelberg, Germany.

[PR18] Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part I*, volume 10991 of *Lecture Notes in Computer Science*, pages 3–32, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[PRSS21] Bertram Poettering, Paul Rösler, Jörg Schwenk, and Douglas Stebila. SoK: Game-based security models for group key exchange. In Kenneth G. Paterson, editor, *Topics in Cryptology – CT-RSA 2021*, volume 12704 of *Lecture Notes in Computer Science*, pages 148–176, Virtual Event, May 17–20, 2021. Springer, Heidelberg, Germany.

[PST23] Kenneth G. Paterson, Matteo Scarlata, and Kien Tuong Truong. Three lessons from threema: Analysis of a secure messenger. In *32nd USENIX*

Security Symposium (USENIX Security 23), pages 1289–1306, Anaheim, CA, August 2023. USENIX Association.

[RAH+16] Scott Ruoti, Jeff Andersen, Scott Heidbrink, Mark O'Neill, Elham Vaziripour, Justin Wu, Daniel Zappala, and Kent Seamons. "we're on the same page": A usability study of secure email using pairs of novice users. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*, CHI'16. ACM, May 2016.

[RAZS16] Scott Ruoti, Jeff Andersen, Daniel Zappala, and Kent Seamons. Why johnny still, still can't encrypt: Evaluating the usability of a modern pgp client. https://arxiv.org/abs/1510.08555, 2016.

[RBH+09] Diego Rybski, Sergey V. Buldyrev, Shlomo Havlin, Fredrik Liljeros, and Hernán A. Makse. Scaling laws of human interaction activity. *Proceedings of the National Academy of Sciences*, 106(31):12640–12645, 2009.

[Rog09] Phillip Rogaway. Practice-Oriented Provable Security and the Social Construction of Cryptography. https://www.cs.ucdavis.edu/~rogaway/papers/cc.pdf, May 2009.

[Rog15] Phillip Rogaway. The moral character of cryptographic work. Cryptology ePrint Archive, Report 2015/1162, 2015. https://eprint.iacr.org/2015/1162.

[Rog19] Phillip Rogaway. An obsession with definitions. In Tanja Lange and Orr Dunkelman, editors, *Progress in Cryptology – LATINCRYPT 2017*, pages 3–20, Cham, 2019. Springer International Publishing.

[Rog24] Phillip Rogaway. Radical CS. https://web.cs.ucdavis.edu/~rogaway/papers/radical.pdf, June 2024.

[Ros22] Leah Namisa Rosenbloom. Activists Want Better, Safer Technology. https://arxiv.org/abs/2209.01273, 2022.

[RS23] Paul Rösler and Jörg Schwenk. Interoperability between messaging services secure implementation of encryption. https://www.bundesnetzagentur.de/DE/Fachthemen/Digitalisierung/Technologien/Onlinekomm/Study_InteropEncryption.html, 04 2023.

[Rus99] Frank Santi Russell. *Information gathering in classical Greece*. University of Michigan Press, 1999.

[Sec24] SecureDrop. Introducing SecureDrop Protocol. https://securedrop.org/news/introducing-securedrop-protocol/, May 2024.

[SH15] Elijah Sparrow and Harry Halpin. *LEAP: The LEAP Encryption Access Project*, pages 367–383. 10 2015.

[Shi11] Clay Shirky. The political power of social media: Technology, the public sphere, and political change. *Foreign Affairs*, 90:28–41, 2011.

[Sig18] Signal. Technology preview: Sealed sender for Signal. https://signal.org/blog/sealed-sender/, October 2018.

[SLI+18] Lucy Simko, Ada Lerner, Samia Ibtasam, Franziska Roesner, and Tadayoshi Kohno. Computer security and privacy for refugees in the united states. *2018 IEEE Symposium on Security and Privacy (SP)*, pages 409–423, 2018.

[SMP24] Yuanming Song, Lenka Mareková, and Kenneth G. Paterson. Cryptographic Analysis of Delta Chat. In *33rd USENIX Security Symposium (USENIX Security 24)*. USENIX Association, aug 2024.

[Spe24] Speedtest. Speedtest Global Index. https://www.speedtest.net/global-index, May 2024. Accessed: 17-06-2024.

[SR16] John Scott-Railton. Security for the high-risk user: Separate and unequal. *IEEE Security & Privacy*, 14:79–87, 03 2016.

[Sta21] Kate Starbird. Online rumors, misinformation and disinformation: The perfect storm of COVID-19 and election2020. In *Enigma 2021*. USENIX Association, February 2021.

[TAB+21] Kurt Thomas, Devdatta Akhawe, Michael Bailey, Dan Boneh, Elie Bursztein, Sunny Consolvo, Nicola Dell, Zakir Durumeric, Patrick Gage Kelley, Deepak Kumar, Damon McCoy, Sarah Meiklejohn, Thomas Ristenpart, and Gianluca Stringhini. SoK: Hate, harassment, and the changing landscape of online abuse. In *2021 IEEE Symposium on Security and Privacy*, pages 247–267, San Francisco, CA, USA, May 24–27, 2021. IEEE Computer Society Press.

[The19] TheCitizenLab. Nso group / q cyber technologies over one hundred new abuse cases. https://web.archive.org/web/20200419152528/https://citizenlab.ca/2019/10/nso-q-cyber-technologies-100-new-abuse-cases/, 2019.

[Tim] The Brussels Times. Belgian EU Presidency widely criticised over 'chat control' vote: What happened? https://www.brusselstimes.com/1102673/belgian-eu-presidency-widely-criticised-over-chat-control-vote-what-happened. Accessed: 08.07.2024.

[Tin20] Tin-Yuet Ting. From 'be water' to 'be fire': Nascent smart mob and networked protests in hong kong. *Social Movement Studies*, 19:362–368, 02 2020.

[Tre20] Emiliano Treré. The banality of whatsapp: On the everyday politics of backstage activism in mexico and spain. *First Monday*, 25, 01 2020.

[Tro23] Carmela Troncoso et al. Joint statement of scientists and researchers on eu's proposed child sexual abuse. https://docs.google.com/document/d/13Aeex72MtFBjKhExRTooVMWN9TC-pbH-5LEaAbMF91Y/edit, 07 2023. Accessed: 17-06-2024.

[URW18] Temple Uwalaka, Scott Rickard, and Jerry Watkins. Mobile social networking applications and the 2012 occupy nigeria protest. *Journal of African Media Studies*, 10:3–19, 03 2018.

[VV17] Augusto Valeriani and Cristian Vaccari. Political talk on mobile instant messaging services: a comparative analysis of germany, italy, and the uk. *Information, Communication & Society*, 21:1–17, 08 2017.

[WGL98] Chung Kei Wong, Mohamed G. Gouda, and Simon S. Lam. Secure group communications using key graphs. In *Proceedings of ACM SIGCOMM*, pages 68–79, Vancouver, BC, Canada, August 31 – September 4, 1998.

[WGL00] Chung Kei Wong, Mohamed Gouda, and Simon S. Lam. Secure group communications using key graphs. *IEEE/ACM Transactions on Networking*, 8(1):16–30, February 2000.

[WHA98] D. M. Wallner, E. J. Harder, and R. C. Agee. Key management for multicast: Issues and architectures. Internet Draft, September 1998. http://www.ietf.org/ID.html.

[WKHB21a] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021: 28th Conference on Computer and Communications Security*, pages 2024–2045, Virtual Event, Republic of Korea, November 15–19, 2021. ACM Press.

[WKHB21b] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*, CCS '21, page 2024–2045, New York, NY, USA, 2021. Association for Computing Machinery.

[WPBB23] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. TreeSync: Authenticated group management for messaging layer security. In *32nd USENIX Security Symposium (USENIX Security 23)*, pages 1217–1233, Anaheim, CA, August 2023. USENIX Association.

[WT99] Alma Whitten and J. Doug Tygar. Why johnny can't encrypt: A usability evaluation of PGP 5.0. In G. Winfield Treese, editor, *USENIX Security 99: 8th USENIX Security Symposium*, Washington, DC, USA, August 23–26, 1999. USENIX Association.

[ZLC17] Hong Zhong, Weiya Luo, and Jie Cui. Multiple multicast group key management for the internet of people. *Concurrency and Computation: Practice and Experience*, 29(3):e3817, 2017. e3817 CPE-15-0502.R1.

[Zuc13] Ethan Zuckerman. Cute Cats to the Rescue? Participatory Media and Political Expression. https://dspace.mit.edu/handle/1721.1/78899, 05 2013.