Cryptographic Foundations of Modern Stateful and Continuous Key Exchange Primitives

Paul Christoph Rösler

Ruhr University Bochum
Horst Görtz Institute for IT-Security
Chair for Network and Data Security

Dissertation zur Erlangung des Grades eines Doktor-Ingenieurs der Fakultät für Elektrotechnik und Informationstechnik an der Ruhr-Universität Bochum

Cryptographic Foundations of Modern Stateful and Continuous Key Exchange Primitives

Paul Christoph Rösler
Place of birth: Arnsberg Neheim, Germany
Email: paul.roesler@rub.de
ORCID: 0000-0002-2324-5671

March 11, 2021
Date of Defense: February 5, 2021

Ruhr University Bochum
Horst Görtz Institute for IT-Security
Chair for Network and Data Security

Dissertation zur Erlangung des Grades eines Doktor-Ingenieurs der Fakultät für Elektrotechnik und Informationstechnik an der Ruhr-Universität Bochum

First Referee: Prof. Dr. rer. nat. Jörg Schwenk
Second Referee: Prof. Dr. phil. nat. Marc Fischlin

nds.rub.de

Acknowledgments

I am convinced that it is very reasonable for researchers to assume full *causality* in this universe. By this, I mean that every event in this universe is the (causal) result of a process that is initiated under certain circumstances. (These circumstances are again events, determined through processes under their influencing circumstances, and so on). Therefore, I believe that I am (only) the result of the influences that let me become who I am today. Since I enjoyed the process of working on this thesis and I am grateful about the event of being rewarded with a Ph.D., I want to thank all my *significant* influences in the following, who constituted the circumstances under which this was possible.

First and foremost, I want to thank my fiance, my family, and my friends for their endless mental support. Thank you, Nadine, for your patience, for being there when I needed you, for leaving me alone when I was busy, for believing in me, for your love, and for planning your future with me. Thank you Mama, Papa, and Karl for laying the foundations of my academic career path, for setting me an example of being interested, curious, enthusiastic, and precise, and for letting me know that I can always rely on you. The success of my research also crucially bases on my prior studies of IT security with the best study group one could imagine. Thank you Dennis, Dennis, Eduard, Fabian, Jan, Jannik, Nils, Philip, and Philipp for jointly exploring adversaries, cryptography, and Bochum's nightlife.

By giving me time, space, a collaborative environment, money, and freedom, Jörg provided everything I needed for using my initial knowledge to investigate cryptographic problems that I found interesting. Thank you very much, Jörg, for letting me do what I considered promising and letting me go where I expected synergies without setting any boundaries. I believe your supervision approach ideally led me to where a Ph.D. student wants to be after their graduation.

I am also thankful to those who connected me to the international cryptography community. Special thanks goes to my second supervisor, Eike, with whom I could discuss future career plans and who

introduced me to my favorite co-author Bertram. Which brings me to Bertram, who taught me so many different lessons during my Ph.D. that I cannot mention all: Thank you, Bertram, for all the discussions about the literature on key exchange and cryptographic modeling in general, for inviting me to London and bringing me into contact with many members of our research community, for teaching me precision and showing me how to avoid arbitrariness, and for being pedantic about how to use LaTeX.

The great research environment that Jörg established was primarily constituted by my colleagues—foremost by my office mate Robert. I am so grateful about all our political, philosophical, and technical discussion that echoed through the hallway such that all colleagues were able to participate passively, about our Friday afternoons during which we reviewed the latest hip-hop releases with a bottle of beer, about our mutual success with which we motivated each other, and I am looking forward to eventually joining RWC in Amsterdam with you next year. My time at the Chair for Network and Data Security wouldn't have started without Christian's and Martin's great supervision of my bachelor thesis after which they invited me to work more closely with this group. During my work, Petra's hidden but significant support helped me to focus on the things I wanted to do. Similarly, by managing our technical and organizational infrastructure, and helping out whenever it was needed, Dennis made my researcher life far easier. Having Sebastian as the second member of our two-persons-theory-minority-subgroup was a real pleasure: I am glad that we jointly organized our (almost) own lecture on key exchange and were able to share and discuss crypto related ideas. I also very much appreciated spending my spare time with Vladislav and Juraj. Beyond learning what being postdoc means in practice and how postdocs plan their careers, I enjoyed solving boulder problems with you instead of thinking about work. Although I do not provide further details, I am equally thankful that I had the opportunity to working with you, Dominik, Jens, Lukas, Marcel, Marcus, Marcus, Matthias, and Simon. In addition to this, I want to thank Pascal and Tibor for our very interesting and pleasant research cooperation.

During my studies, I was fortunate to being invited by Bertram, Serge, Kenny, and Yevgeniy to their groups for collaborations. Every one of you was a great host to me and I am so glad that I had the chance to think, discuss, and do research with you. For doing joint research, structuring work, formulating results, and exploring cryptography, I am deeply thankful to all of my co-authors, especially Alexander, Benjamin, and Fatih, who expended enormous efforts with me to finalize our papers. Last but not least, I want to thank Marc for accepting to review my thesis and assessing my defense. I am very happy that I am able to work with you now.

Abstract

Protecting communication cryptographically is usually considered and realized modularly: Protocols for protecting the actual payload utilize session keys that are exchanged by other (independent) protocols. *Key exchange protocols*, securely establishing session keys between authorized parties such that outsiders cannot compute these keys, are one of the most fundamental primitives in cryptography. We consider in this thesis modern key exchange protocols that *continuously* establish new session keys for involved parties. Furthermore, we concentrate on techniques, often implemented in these protocols, that steadily update secrets used by the involved parties while establishing session keys. These update mechanisms are applied to protect session keys against adversaries who can temporarily expose parties' local state memory on which these secrets are stored. Realistic examples that illustrate this adversarial capability include physical access to victims' devices, computer viruses on these devices, implementation flaws in the affected software, etc. For updating the current state secrets of parties, freshly generated secrets are steadily mixed into the state, and old secrets are simultaneously erased from it. These update mechanisms, metaphorically called *ratcheting*, were developed for modern messaging protocols and are today deployed in all relevant messaging applications; Similar techniques are also implemented in traditional group key exchange protocols. In order to provide methodical and systematic foundations that facilitate access to these mechanisms for academic research, we conceptually and fundamentally rethink their modeling and realization.

With our consideration of two-party ratcheting (i.e., continuous key exchange under steady updates of secrets), we are the first to propose a natural security definition that takes realistic interaction between parties into account. We divide our notions of interaction into three incremental stages to reduce complexity due to entailed concurrent and asynchronous communication. Thereby, our results reveal core elements of ratcheting in our definitions as well as in our accordingly secure constructions.

Some of these constructions are built from remarkably strong cryptographic building blocks. In our analysis of reasons for this, we prove that these strong building blocks are under practical conditions necessary to build ratcheting, crystallizing them as core components thereof. As part of this analysis, we introduce a natural security notion of ratcheting that takes the vulnerability of utilized random coins appropriately into account.

We subsequently turn to concurrency as a crucial problem of ratcheting in groups. More precisely, we analyze the communication overhead that results from multiple group members concurrently updating the secrets in their local states. Previous group ratcheting protocols only allow for sequential state updates, causing a significant efficiency loss for practical deployment. We prove almost tight lower and upper bounds of communication complexity. In doing so, we reveal theoretical performance limits and provide a practical scheme for concurrent group ratcheting.

Finally, we more generally examine modeling for group key exchange. While group key exchange has a long tradition in cryptography, its modeling also becomes increasingly relevant due to the advent of group ratcheting. We systematically review, compare, and evaluate all relevant models in the literature. In order to resolve correspondingly detected issues and shortcomings, we propose a simple and generic model, which may serve as a foundation for future modeling attempts.

Conclusively, the results of this thesis comprise systematic models and constructions that are directly related to modern means of communication. Furthermore, we reveal relations between various cryptographic primitives and determine corresponding performance limitations. Various methodologies and solutions developed in this thesis (re-)structure the current state of research on continuous and stateful key exchange. Due to their generic nature, many of our results can be useful in applications and extensions far beyond the specific problems, covered in this thesis.

Zusammenfassung

Geschützte Kommunikation wird in der Kryptographie üblicherweise modular betrachtet und realisiert: Die Protokolle zur Sicherung der eigentlichen Kommunikation verwenden Sitzungsschlüssel, die von anderen (unabhängigen) Protokollen ausgetauscht werden. *Schlüsselaustauschprotokolle*, die das sichere Austauschen von Sitzungsschlüsseln zwischen autorisierten Parteien ermöglichen, sodass außenstehende Parteien diese Schlüssel nicht berechnen können, sind eine der fundamentalsten Primitiven in der Kryptographie. Im Fokus der Betrachtung dieser Dissertation liegen sowohl moderne Schlüsselaustauschprotokolle, die *kontinuierlich* neue Sitzungsschlüssel zwischen involvierten Parteien austauschen, als auch darin häufig verbaute Techniken, die die dafür verwendeten Geheimnisse der Parteien stetig erneuern. Dieses Erneuern der Geheimnisse soll ausgetauschte Sitzungsschlüssel vor Angriffen schützen, bei denen Angreifern zeitlich beschränkter Zugriff auf den Zustandsspeicher der Parteien, der jene verwendete Geheimnisse enthält, möglich ist. Realistische Beispiele, die diese Angriffsfähigkeit veranschaulichen, umfassen physischen Zugriff auf Geräte der Opfer, Computerviren auf diesen Geräten, Implementierungsfehler in betroffener Software, etc. Für das Erneuern werden stetig neu generierte Geheimnisse in den aktuellen Zustand gemischt und gleichzeitig alte Geheimnisse daraus entfernt. Diese Erneuerungstechniken, die metaphorisch *Ratcheting* (engl. 'Ratcheting' = 'Ratschen') genannt werden, wurden für moderne Messengerprotokolle entwickelt und sind in fast allen aktuell relevanten Messengerdiensten verbaut; Ähnliche Mechanismen finden sich darüber hinaus auch in traditionellen Gruppenschlüsselaustauschverfahren. Um der Forschung methodischen und systematischen Zugang zu diesen Techniken zu verschaffen, konzentrieren sich die Arbeiten im Rahmen dieser Dissertation darauf, sie konzeptionell und grundsätzlich neu zu erfassen, zu modellieren und zu realisieren.

Für das zunächst betrachtete Zwei-Parteien-Ratcheting—also dem kontinuierlichen Austausch von Sitzungsschlüsseln bei gleichzeitig stetigem Erneuern verwendeter Geheimnisse—wird in der vorliegenden

Arbeit die erste natürliche Sicherheitsdefinition vorgestellt. Diese Definition trägt in drei aufeinanderfolgenden Stufen realistischer Interaktivität, und damit simultaner und asynchroner Kommunikation, zwischen den involvierten Parteien Rechnung. Die stufenweise Betrachtung der Interaktivität ermöglicht zum einen eine klarere Darstellung der Ergebnisse und deckt zum anderen wesentliche Bestandteile des Ratchetings in den Definitionen und auch in den vorgelegten, entsprechend sicheren, Konstruktionen auf.

Darauffolgend wird die Verwendung auffällig starker kryptographischer Komponenten in diesen Konstruktionen analysiert. Es werden praktische Bedingungen vorgestellt, unter denen diese mächtigen Komponenten beweisbar notwendig sind um Ratcheting zu konstruieren, wodurch sich diese Komponenten als Kernstück natürlich sicheren Ratchetings herauskristallisieren. Im Rahmen dieser Analyse wird außerdem eine natürliche Sicherheitsdefinition für Ratcheting eingeführt, die die Schwächung verwendeter Zufallszahlen angemessen berücksichtigt.

Anschließend wird ein wesentliches Problem für Ratcheting in Gruppen betrachtet: es wird der Kommunikationsaufwand untersucht, der verursacht wird, wenn mehrere Gruppenmitglieder gleichzeitig Geheimnisse erneuern. Bisherige Verfahren ermöglichen nur sequenzielles Erneuern von Geheimnissen, was zu signifikanten Effizienzeinbußen in der Praxis führt. Es werden enge untere und obere Schranken des verursachten Kommunikationsaufwands bewiesen, woraus zum einen theoretische Grenzen der Performanz und zum anderen ein praktisches Verfahren für Gruppenratcheting hervorgehen.

Genereller wird abschließend die Modellierung von Gruppenschlüsselaustauschverfahren betrachtet, die eine lange Tradition in der Kryptographie hat, aber auch aktuell Relevanz durch das aufkommende Gruppenratcheting erfährt. Systematisch werden Eigenschaften aller relevanten Modelle in der Literatur begutachtet und verglichen. Um dabei offengelegte Probleme und Unzulänglichkeiten zu überwinden, wird ein simples, generisches Modell vorgeschlagen, womit zukünftiger Forschung eine Modellierungsgrundlage geboten wird.

Die Ergebnisse dieser Dissertation umfassen dementsprechend neue,

systematische Modellierungen und Konstruktionen, die direkten Bezug zu modernen Kommunikationsmedien haben. Des Weiteren werden Relationen zwischen verschiedenen kryptographischen Primitiven hergestellt, sowie Effizienzschranken festgestellt. Viele der entwickelten Methoden und Lösungen (re-)strukturieren den aktuellen Forschungsstand zu kontinuierlichem und zustandsbehaftetem Schlüsselaustausch. Durch ihren generischen Charakter können diese Ergebnisse weit über die speziellen Probleme, die diese Dissertation behandelt, Anwendung und Erweiterung finden.

Contents

1 Introduction **1**
- 1.1 Secure Messaging between Endpoints . . . 3
- 1.2 Ratcheting . . . 7
- 1.3 The Group Setting . . . 12
- 1.4 Further Contributions . . . 16
- 1.5 Organization . . . 19

2 Preliminaries **21**
- 2.1 Cryptographic Modeling . . . 21
- 2.2 Notation . . . 24
- 2.3 Cryptographic Building Blocks . . . 26

3 Optimally Secure Ratcheting in Two-Party Settings **39**
- 3.1 Introduction . . . 41
- 3.2 Key-updatable Key Encapsulation Mechanisms . . . 47
- 3.3 Unidirectionally ratcheted key exchange (URKE) . . . 51
- 3.4 Constructing URKE . . . 57
- 3.5 Sesquidirectionally ratcheted key exchange (SRKE) . . . 60
- 3.6 Constructing SRKE . . . 66
- 3.7 Rationales for SRKE Design . . . 72
- 3.8 Bidirectionally ratcheted key exchange (BRKE) . . . 78
- 3.9 Constructing BRKE . . . 80
- 3.10 Proof of URKE . . . 86
- 3.11 Proof of SRKE . . . 95
- 3.12 Proof of BRKE . . . 113
- 3.13 Modeling ratcheted key exchange . . . 117

4 Necessity of Strong Building Blocks for Optimally Secure Ratcheting **121**
- 4.1 Introduction . . . 122
- 4.2 Sufficient Security for Key-Updatable KEM . . . 130
- 4.3 Unidirectional RKE under Randomness Manipulation 138

4.4 kuKEM* to URKE 144
4.5 URKE to kuKEM* 149
4.6 Discussion 161

5 Communication Costs of Ratcheting in Groups 167
5.1 Introduction 168
5.2 Security of Concurrent Group Ratcheting 176
5.3 Deficiencies of Existing Protocols 179
5.4 Key-Updatable Public Key Encryption 185
5.5 Intuition for Lower Bound 186
5.6 Upper Bound of Communication Complexity 194
5.7 Lower Bound of Communication Complexity 202
5.8 Discussion 227

6 Systematization of Models for Key Exchange in Groups 231
6.1 Introduction 232
6.2 Syntax Definitions 237
6.3 Communication Models 248
6.4 Security Definitions 262
6.5 Discussion 271

7 Conclusions and Outlook 275
7.1 Overview 275
7.2 Statefulness 278
7.3 Defining Syntax, Correctness, and Security 279
7.4 Continuous State Updates 280
7.5 Asynchronicity 282
7.6 Two Perspectives on Problems 283
7.7 Impact 284

1 Introduction

Our research on the cryptographic primitives, covered in this thesis, is substantially motivated by their use in modern real-world messaging applications. The common technique that all these primitives implement are mechanisms to continuously update employed key material. We consider these key-updating mechanisms from various perspectives in order to derive a clearer, systematic understanding thereof. Before switching to a more research-oriented and technical perspective in this introduction, we will first take a look at the short history of modern secure messaging.

With the market introduction of smartphones and the emerging accessibility of mobile Internet, the advent of messaging applications was initiated in the late 2000s. For example, WhatsApp, being the most widely used messaging application worldwide today[1], was launched in 2009.[2] Less used, alternative messengers like TextSecure from early on incorporated into their consideration adversaries and, therefore, implemented encryption mechanisms. Conceptually, the first available version of TextSecure from 2011[3] encrypted payload messages on the sender device, transmitted the resulting ciphertexts via SMS to the receiver, and let the receiver decrypt the ciphertexts to obtain the sent messages. Many messengers like Telegram or Threema followed this *end-to-end* encryption concept.

An early feature of TextSecure was implementing continuous key-

[1] https://www.statista.com/statistics/258749/most-popular-global-mobile-messenger-apps/

[2] WhatsApp 2.0 announcement from 2009-08-27: https://blog.whatsapp.com/whats-app-2-0-is-submitted

[3] Initial commit from 2011-12-20: https://github.com/signalapp/Signal-Android/tree/bbea3fe1b110afadbac4d6a2c183dc4d899e4ead

update mechanisms, adopted from the off-the-record (OTR) messaging protocol[4], to change locally stored encryption keys of communication partners irreversibly. With this technique, adversarial exposures of this key material (e.g., due to physical access to user devices) remain harmless to the confidentiality of *previously* communicated messages. The update mechanism also renewed the key material of communication participants by mixing secretly generated fresh values into it. This mixing process implies that also confidentiality of *future* messages is regained after an adversarial exposure of stored keys. TextSecure, which was later renamed to Signal, enhanced OTR's key-update mechanism further[5] and made it known under the term 'ratcheting'. This mechanism was eventually even integrated into WhatsApp.[6]

Key updates to protect against previous and later threats are, however, neither new nor exclusive to modern (two-party) messaging. For example, traditional group key agreement in dynamic groups requires that only the current set of group members can compute an established key, whereas potentially malicious previous and future members are unable to do so.

After these key-updating techniques were implemented in practice and partially considered by academic research before, our approach in this thesis is to rethink how their abstract concepts can be modeled by cryptographic primitives generically. We thereby also take into account today's user demands, extended technical opportunities, restrictions in realistic deployment settings, and many other special characteristics of modern messaging applications that affect these cryptographic primitives. By using well-established methodologies, we review and systematize previous definitional approaches, and introduce realistically stronger and more natural (security) definitions. Furthermore, we analyze qualities and implications of these definitions

[4] Protocol description from 2012-10-18: https://github.com/signalapp/Signal-Android/wiki/Protocol/cfdfe3bbae7a06cd1cc6f1aee5bdf918b86b58d8

[5] Announcement from 2013-09-26 https://signal.org/blog/advanced-ratcheting/

[6] Blog post from 2014-11-18: https://signal.org/blog/whatsapp/

as well as relations between them. Finally, we propose accordingly secure constructions. Although our motivation is based on modern messaging, our results are due to their generality applicable beyond this particular use case.

Below, we point out challenges and crucial requirements of secure messaging that we take into account, introduce relevant methodical strategies and influential related literature that our work bases on, and establish important abstract terms that we use, in order to comprehensibly present our own contributions embedded in the field of cryptography.

1.1 Secure Messaging between Endpoints

Of the long history of secure digital user-to-user communication, only one legacy protocol is still considerably deployed and used: End-to-end encrypted email via PGP or S/MIME.[7] Abstractly, the concept behind this technology is stateless public key encryption: the sender encrypts the payload of an email to the static public key of the receiver. Although many recent articles reveal serious vulnerabilities [PDM+18, MBP+19a, MBP+19b, MBP+20], the main features of email with these piggyback protection mechanisms appear to remain decisive: wide deployment of email, ad-hoc nature of communication, and publicly accessible contact information.

Mobile Devices and Centralized Networks. In contrast to email, which is traditionally used on desktop computers, modern messaging applications are primarily designed for mobile use on smartphones. As a consequence, the local network of messengers constantly changes and is typically not under control of the user. Furthermore, while email is a (more or less)[8] decentralized protocol, all established

[7] Use of email is still growing: https://www.statista.com/statistics/255080/number-of-e-mail-users-worldwide/; Poddebniak et al. [PDM+18] list various human rights organizations that recommend the use of PGP and S/MIME to protect emails.

[8] Gmail had more than 1.5 billion active users in 2018 and each email to or

messaging applications mandatorily route every communication via their respective single central service provider. Assuming that the presence of adversaries is more likely in partially uncontrollable networks, and anticipating that the center of a network is furthermore a high-value target, it seems reasonable to consider communication via messaging applications vulnerable. Hence, demanding protection against network adversaries from these applications seems justified. In fact, all major messaging applications used in liberal countries (i.e., WhatsApp, Facebook Messenger, and Telegram)[1] implement end-to-end encryption methods to protect communication.

Additionally and independent of network attackers, we emphasize that the mobile use of messengers, compared to software run on stationary home computers, increases the risk that adversaries obtain physical access to the local user device, exposing all secrets stored on it.

Statefulness and Long-term Sessions. In contrast to encrypted email where senders encrypt each message individually with the static public key of receivers, modern messengers can use the local device memory of smartphones to store, reuse, and update key material. As we show in a work that is independent of this thesis [RMS18, Rös18], all considered messenger protocols indeed utilize the local state to protect communication, which we call *statefulness*. For example, Threema, at the point of our analysis, let communication participants initiate a conversation by establishing a symmetric key from the Diffie–Hellman key exchange of their long-term Diffie–Hellman shares. This symmetric key was then permanently stored in the participants' local states and used to encrypt all their communicated messages.

Due to statefulness, messaging sessions *comprise* the entire conversation between participants until one of them discards their local session state. As the latter is usually linked to changing the device, messaging sessions may take years. Adversaries, obtaining access to a local session state—which can be more likely due to the mobile use of

from these users is delivered via Google servers: https://www.statista.com/statistics/432390/active-gmail-users/

messaging apps—, hence, potentially *compromise* the security of all conversations processed during the lifetime of a smartphone. While this would have been the case in the above-given example of Threema, we will point out below that using statefulness *properly* easily enables significantly strengthened security, opposing this increased risk.

Asynchronicity. The use of messengers on *mobile* smartphones has, beyond the higher risk of state exposures, the effect that communication participants are never guaranteed to be or remain online, even during an established conversation. However, users want to send protected messages at any time, independent of their counterpart's current availability. This requirement includes the ability to initiate a conversation non-interactively. The receiving counterparts should be able to process incoming messages and respond as soon as they are online, independent of whether the original sender is still available. This asynchronous communication setting also requires that all participants should be able to simultaneously and concurrently contribute to the communication without being forced to follow a synchronized schedule. In other words, sending messages should not depend on communication partners' current status, and messages may 'cross' on the wire.

Forward-Secrecy. In addition to physical adversarial access to user devices, many other examples highlight that exposures of locally stored user secrets are a practical threat: viruses can steal secrets until they are eliminated, remote backups or log files may unintentionally include key material, temporary implementation bugs in messenger applications may reveal their secrets, attackers can use cryptanalysis against single keys, law enforcement agencies may lawfully coerce users to reveal their device's memory, etc.

The abstract security goal *forward-secrecy* requires that constructions implement measures to guarantee security of *past* communication after an involved participant's local state was exposed to an adversary. This goal is both acknowledged and well established in modern cryptography. However, under all above-mentioned circumstances, protecting the security of communication under exposures of users' local

state secrets is a highly complicated problem:

A classic first-key-agreement-then-symmetric-protocol approach for achieving forward-secrecy in the key-agreement phase usually involves interaction, for example, when employing a Diffie–Hellman key exchange (DHKE) with ephemeral key material. As mentioned earlier, interaction between participants during the initialization of a conversation cannot be guaranteed. Recall that Threema, for avoiding interaction, initiated conversations with a key exchange under participants' publicly distributed *static long-term* key material and, hence, failed to reach forward-secrecy.

Recent literature on forward-secure key exchange [GHJL17, DJSS18] proposes yet inefficient non-interactive mechanisms. Practical protocols, instead, circumvent the interaction-barrier efficiently by utilizing central key servers with which users emulate interaction: they proactively prepare multiple ephemeral contributions to the interactive key exchange and store them on the server such that their counterparts can download one contribution each and react without relying on simultaneous activity [CCD+17].

Strong forward-secrecy goes beyond the properties of session initialization: if session secrets are initially forward-securely established but remain static afterwards, the security gain for long-term sessions under state exposures is negligible. Consequently, messaging applications are today required to implement forward-secure continuous key updates mechanisms within established conversations.

Recovery from Exposures. If an exposure happens early in a long conversation, the impact of this attack is severe even if forward-secrecy is maintained because messages sent thereafter are not protected by it. The opposite security goal, *post-compromise security* [CCG16][9], requires that a protocol continuously 'heals' its users' local state secrets such that the effect of their exposure on *future* communication's security is limited. Modern messaging protocols simultaneously achieve forward-secrecy (FS) and post-compromise security (PCS) to minimize the compromise caused by state exposures. We note that this ter-

[9] 'Future secrecy' and 'backward secrecy' used to be synonyms.

minology (i.e., 'FS' and 'PCS') only refers to conceptual ideas rather than clearly defined security goals.

Randomness Sources. Fresh randomness is a crucial ingredient to achieve post-compromise security: If an adversary temporarily obtained all secrets of a party, the only thing that this party can do to regain a secure conversation after this exposure is to generate new key material from secretly sampled random coins. As a result, reliance on 'good' randomness and security even under attacked randomness are aspects that security of modern messengers is also confronted with.

1.2 Ratcheting

One technique implemented by messengers to continuously update session states is via 'hash chains' where the symmetric key material contained in the local state is replaced, after each use, by a new value derived from the old value by applying some one-way function. This method mainly targets forward-secrecy and has a long tradition in cryptography. A second technique is to let participants routinely redo a DHKE and mix the newly established keys into the session state: As part of every outgoing message, a fresh g^x value is combined with prior and later values g^y contributed by the peer, with the goal of refreshing the session state as often as possible. This was introduced with the OTR messaging protocol from [OTR16, BGB04] and promises auto-healing after a state exposure, at least if the DHKE exponents are derived from fresh randomness gathered from an uncorrupted source after the state exposure took place. Of course, the two methods are not mutually exclusive but can be combined.

We say that a messaging protocol employs *ratcheting* if it uses the described or similar techniques for achieving forward-secrecy and post-compromise security. In this context the term 'ratcheting' can be traced back to the Pond protocol [Lan16] and refers to the idea that new key material is continuously input into the current state, and old material in the state is continuously discarded such that this process cannot be reversed (forward-secrecy) and future states cannot be

foreseen (post-compromise security).

Unless explicitly stated otherwise, we consider ratcheting protocols in this thesis modularly in the sense that they consist of a mechanism that continuously establishes symmetric keys and a cleanly separated channel that uses these keys to protect payload. Since adding a channel on top of the key establishment component is often straightforward, we almost exclusively focus on the establishment of keys here.

From Design Technique to Primitive Class. The most prominent and most widely deployed real-world ratcheting protocol is the Double Ratchet Algorithm (sometimes called Signal Protocol) [PM16]. This protocol by Marlinspike and Perrin is used by all major messaging apps (the Signal Messenger, WhatsApp, Facebook Messenger, Skype, and others). Most importantly, it initiated the academic research on practical ratcheting constructions and theoretical aspects thereof.

An influential milestone for the academic consideration of ratcheting was the seminal work by Bellare et al. [BSJ+17]. Instead of following the construction-driven perspective, they abstractly capture the concept of *ratcheted key exchange* (RKE) by defining a suitable, generic syntax, and thereby considering it a distinct cryptographic primitive. To focus on the core idea of ratcheting, they disregard extraneous components like the initialization of a session or the confidential messaging channel. Furthermore, they concentrate on only the Alice-to-Bob direction of the communication. In this syntax they define three algorithms: a non-interactive initialization algorithm with which Alice and Bob jointly compute their local session states, a send algorithm that can be invoked by Alice, and a receive algorithm that can be invoked by Bob. With the latter two, Alice and Bob can continuously establish symmetric keys in the session.

Alice uses her local state in the send algorithm to compute a symmetric key, derive an according ciphertext, and update her state. When obtaining a ciphertext from Alice, Bob can use his state in the receive algorithm to compute (the same) symmetric key and update his state. Since Alice and Bob cannot switch roles in this syntax notion (i.e., Bob can never respond), we refer to it as *unidirectional*

RKE. With this clear concept, Bellare et al. abstractly cover the interfaces of ratcheting algorithms to their environment, and thereby divide ratcheting from the established key agreement[10] literature that often dispenses with precise syntax specifications (see Paragraph Definitions below).

Defining Security Methodically. In addition to defining a distinct syntax, Bellare et al. apply a well-established methodology in cryptography on this primitive to obtain an *almost* natural notion of security: they define a game played by an adversary $\mathcal{A}$ in which the execution of an RKE session is simulated. In this game, $\mathcal{A}$ can let Alice and Bob invoke their respective algorithms and determine these invocations' public input parameters. Furthermore, adversary $\mathcal{A}$ can expose Alice's internal local state secrets. Winning the game is conditioned on $\mathcal{A}$ being able to distinguish keys, established between Alice and Bob during the game, from randomly sampled keys. As the described adversarial power allows for trivially and unpreventably solving some game challenges (e.g., keys computed by Bob after an adversarial impersonation of Alice), the adversary is restricted to solving only those challenges that can theoretically be protected by an RKE construction.

The main novelty here is the minimal restrictions of adversaries in the last mentioned step: this approach defines security of RKE via determining what theoretically *could* and potentially *should* be achieved by constructions—which we call *naturally* as it follows a clear methodology that is standard for many cryptographic primitives like security under *chosen ciphertext attacks* (CCA) for encryption [RS92][11]—rather than analytically determining the notion security from what one particular protocol actually achieves.

[10] Please note our distinction between *key agreement* and *ratcheted key exchange* protocols. The former is run by parties who share no common secrets to obtain a symmetric key for initiating subsequent session protocol. The latter is the session protocol that might utilize the initial key and that continuously outputs symmetric keys in the session independent of long-term keys.

[11] Unfortunately, even CCA-security is slightly ambiguous [BHK15] even beyond the obscure lunchtime security notion [NY90].

Considering Bidirectional Interaction. We mention these technical details here already to make the reader familiar with methodical approaches of defining syntax and security that are important throughout this entire work. Turning to the more abstract view, the first heavy restriction of the above presented definitional basis by Bellare et al. is that adversaries are unrealistically forbidden to expose Bob's local state; This results in unnecessarily relaxed forward-secrecy requirements. A far more serious limitation is the restricted communication setting to only *unidirectional* interaction between Alice and Bob.

In Chapter 3 we present our results based on article [PR18b] from CRYPTO 2018 in which we overcome these issues by first redefining security of *unidirectional* RKE and then extending syntax and security concerning permitted user interaction in two steps: We introduce the notion of *sesquidirectional* (lat. 'sesqui-' means 'one and a half') RKE that allows Bob to send 'non-functional' ciphertexts back to Alice. By allowing Bob to contribute information, these ciphertexts can strengthen theoretically achievable security but do not initiate the (functional) establishment of symmetric keys in the Bob-to-Alice direction. Based on this, we formulate the notion of fully *bidirectional* RKE in which both parties equally participate in the continuous establishment of keys. Due to the fully asynchronous and, hence, concurrent interaction in the bidirectional setting, the corresponding security notions can be considered an order of magnitude more complex than those from the unidirectional setting. Our staged approach, consequently, allows us to substantially reduce complexity and clearly point out the core elements of ratcheting.

Constructions. For all our security notions we provide constructions and accordingly prove their security in Chapter 3. Similar to the complexity of security definitions, the corresponding ratcheting constructions become increasingly complex for the bidirectional variants. However, our incremental consideration of interaction pays off: we can generically build bidirectional RKE from two sesquidirectional RKE schemes. Nevertheless, while our unidirectional RKE construc-

tion generalizes previous ratcheting designs and uses standard building blocks such as public key encryption, we introduce and use new *key-updatable* public key encryption (kuPKE) primitives to instantiate our sequidirectional RKE notion. Intuitively, kuPKE is standard public key encryption of which public key and secret key can be updated independently such that differing updates lead to incompatible keys.

Interestingly, we build this new key-updatable primitive from the powerful and sophisticated tool *hierarchical identity-based encryption*. Hence, we use far stronger and less efficient tools for sesqui- and bidirectional RKE than needed for our unidirectional notion and implemented in the practical Signal protocol.

[PR18b] Bertram Poettering and **Paul Rösler**. Towards bidirectional ratcheted key exchange. In Advances in Cryptology – CRYPTO 2018

All formal results as well as large parts of the textual descriptions in this work except for the definitions were contributed by the author of this thesis.

Efficiency. The just highlighted intuitive separation between unidirectional RKE and bidirectional RKE with respect to the hardness of their constructions' building blocks, revealed by the results in Chapter 3, leads us to the following questions: 1. Is strong key-updatable public key cryptography necessary to realize naturally secure ratcheting? 2. If yes, under which conditions can this necessity be proven? 3. Under which conditions is standard public key cryptography sufficient to realize ratcheting?

Motivated by these questions, we investigate the relation between ratcheting and key-updatable public key encryption in Chapter 4 that is based on our article [BRV20a] from ASIACRYPT 2020. Our surprising result is that already unidirectional RKE necessarily relies on building blocks as hard as kuPKE under slightly adapted, realistic conditions.

In order to prove this, we consider attacks against randomness in ratcheting. We thereby introduce the first *natural* security definition

of unidirectional RKE that takes into account manipulation of random coins and exposures of participants' local states in a unified form.

With these results we determine key-updatable public key encryption as the core primitive of strongly secure ratcheting such that future research can concentrate on enhancing constructions thereof. Furthermore, we methodically point out under which conditions ratcheting can be built from less expensive components. Finally, we provide the first clear security notion of ratcheting under attacked randomness.

[BRV20a] Fatih Balli, **Paul Rösler**, and Serge Vaudenay. Determining the core primitive for optimally secure ratcheting. In Advances in Cryptology – ASIACRYPT 2020
This entire work except for one proof and minor textual revisions was contributed by the author of this thesis.

1.3 The Group Setting

Beyond two-party messaging, conversations in groups are an important, much-used feature of messaging applications.[12] For accordingly lifting ratcheting concepts to group conversations, many obstacles arise for which solutions in the two-party setting are insufficient or do not even exist. Before turning to constructions and problems thereof, we discuss differences between these settings on a conceptional level.

Larger Attack Surface. Conversations with more participants are inherently more susceptible to state exposures because their attack surface increases linearly. Moreover, while forward-secrecy can generally be achieved by locally processed updates of the key material without relying on communication between the users, achieving post-compromise security crucially relies on interactive (update) contributions from the affected participants. Hence, the local state of the least active participant in a group poses the greatest security risk regarding state exposures. Maintaining a group conversation secure, con-

[12] https://www.statista.com/statistics/800650/group-chat-functions-age-use-text-online-messaging-apps/

sequently, requires all participants to regularly and frequently share new information with the group.

Complex Interaction. Not only because it is supportive for gaining security, interaction within groups is intensified and thereby more complicated. The probability of simultaneous and, hence, concurrent contributions increases quadratic in the number of members if sending is uniformly distributed. If updating the local state and sharing according information is linked to the transmission of payload in groups, concurrency is almost unavoidable. Consider, for example, one user posting a question to a group chat. If multiple users are online on receipt and formulate their reply immediately, these replies are likely (also due to network latency) sent and processed concurrently.[13] Although concurrency is not an issue per se, techniques from the two-party setting do not directly scale for processing and 'merging' concurrently contributed update information of arbitrarily many users: The above sketched simple state update mechanism based on Diffie–Hellman key exchange works perfectly under concurrency for two-party chats, but a practical group equivalent (i.e., multi-party non-interactive key exchange) is not available.

Constructions and Standardization. Abstractly, ratcheting in groups captures the idea of continuously establishing key material between many users under adversarial state exposures. This idea is conceptually already covered by traditional dynamic group key agreement: The exposed state secrets of a user Alice can be thought of as her current identity being a malicious member in a group. For contributing new key material to recover from this exposure, Alice can remove this malicious member (i.e., her own identity) from the group and immediately add a freshly generated identity to the group again.

In a work independent of this thesis, we analyze how group chats are actually implemented in popular real-world messenger applica-

[13] Delivery acknowledgments automatically inform the sender of a message that the respective receiver obtained this message. If sending these acknowledgments initiates state updates on the receiver side, as implemented in Signal [Rös19], concurrency is provoked for every payload message in a group.

tions [RMS18, Rös18]. Among other things, we show that Signal's approach is to simply send group messages via the pairwise (post-compromise secure) channels between the sender and every other group member. While this mechanism bypasses concurrency problems, it induces a communication overhead per (state update) message linear in the number of group members.

Many recent group ratcheting schemes were developed based on the above-sketched concept of updating user states via dynamic membership replacements using group key agreement techniques. With this approach, a far better, only logarithmic, communication overhead is achieved, but none of these schemes can appropriately handle concurrency.

Concurrency, Quick Recovery, and Overhead. For processing concurrently sent ciphertexts with which their senders aim to update their local state in a group, all previous group ratcheting constructions implement either of the following three alternatives: 1. Receivers reject all but one of the sent ciphertexts such that only one sender achieves post-compromise security for their local state; 2. Receivers merge the sent ciphertexts in a way that maintains a logarithmic communication overhead but is ineffective for gaining post-compromise security; 3. Receivers give up on the logarithmic communication overhead and fall back to a parallel execution of pairwise channels; They potentially regain better efficiency as soon as the senders sent again non-concurrently.

In Chapter 5 that bases on our article [BDR20b] from TCC 2020 we analyze whether this apparent tension between post-compromise security, concurrency, and communication overhead is inherent. Thereby we search for the minimal overhead under concurrency for achieving post-compromise security in groups. By providing both a lower and upper bound of communication complexity in this setting, we answer these questions almost conclusively.

For our lower bound, we develop a symbolic execution model with which we capture the minimal setting in which this tension already occurs, being most restrictive for adversaries and both least demand-

ing as well as most liberal for constructions. Our proof shows that a communication overhead linear in the number of concurrently active group members is inevitable to achieve post-compromise security in this setting. We complement this lower bound with a group ratcheting construction that achieves a communication overhead that is almost tight to it (i.e., up to a logarithmic factor).

[BDR20b] Alexander Bienstock, Yevgeniy Dodis, and **Paul Rösler**. On the price of concurrency in group ratcheting protocols. In Theory of Cryptography – TCC 2020

All formal results as well as the majority of textual descriptions in this work except for the protocol construction and two proofs for the upper bound were contributed by the author of this thesis.

Definitions. Besides construction ideas, group key agreement and the key exchange-core of group ratcheting are similar in the abstract and generic intuitions of their (user-)interfaces, adversarial threats, and security requirements. Hence, it is not surprising that definitional approaches of the emerging group ratcheting research partially adopt and inherit established definitions from the long history of group key agreement literature.

In order to understand relations between, discover advantages and disadvantages of, and extract recommendations on how to design, security models for these cryptographic primitives, we systematize this literature in Chapter 6 that bases on our work [PRSS21] from CT-RSA 2021. Although it is clear that generality in security models is necessary to allow for comparability of constructions, our systematization confirms the common intuition in the research community that (group) key agreement models are usually defined for the single purpose of analyzing one particular protocol. We therefore augment our systematizing analysis with the design of a model that, based on lessons learned from the literature but also crucial reconsiderations of modeling approaches therein, neglects extraneous elements and aims for simplicity and generality.

With these results we facilitate access to the considered literature

and hope to inspire future work on group ratcheting such that security models become more general and comparable.

[PRSS21] Bertram Poettering, **Paul Rösler**, Jörg Schwenk, and Douglas Stebila. SoK: Game-based Security Models for Group Key Exchange. In Topics in Cryptology – CT-RSA 2021
The entire systematization as well the new model design were almost exclusively contributed by the author of this thesis. This work was supported by fruitful discussions with, and textual revisions by, the co-authors.

1.4 Further Contributions

In addition to the work covered in this thesis, we contributed to further topics in IT-security and cryptography. The exact contributions to the following chapters' contents are explicitly declared at the beginning of each chapter, if necessary.

Lead Author The author of this thesis is the lead author of the articles incorporated in this work as well as the already mentioned analysis of group chat implementations in real-world messaging applications [RMS18] that has already been part of his Master's thesis [Rös18].

[RMS18] **Paul Rösler**, Christian Mainka, and Jörg Schwenk. More is less: On the end-to-end security of group chats in Signal, WhatsApp, and Threema. In IEEE European Symposium on Security and Privacy – EuroS&P 2018 (authors ordered by their contributions)
All formal results as well as the majority of textual descriptions in this work were contributed by the author of this thesis.

[Rös18] **Paul Rösler**. On the end-to-end security of group chats in instant messaging protocols. Master's thesis, Ruhr University Bochum, 2018

In another more recent work [DRS20], the author of this thesis, again in the role of the lead author, developed a new security model for modern special-purpose channel establishment protocols. Our more abstract modeling approach in this article was necessary to formally analyze such protocols' security because previous models were unable to cover integrated continuous key update mechanisms, flexibly adaptable security guarantees, and a lack of modularity. We use this model in our security analysis of protocols from the Noise framework [Per17]. This analysis is particularly important because this framework is implemented in widely deployed applications like WhatsApp.

[DRS20] Benjamin Dowling, **Paul Rösler**, and Jörg Schwenk. Flexible authenticated and confidential channel establishment (fACCE): Analyzing the Noise protocol framework. In Public-Key Cryptography – PKC 2020
The entire model design as well as the majority of textual descriptions were contributed by the author of this thesis.

Furthermore, the author of this thesis was the lead author of an article that proposes constructions and analyzes properties of combiners for authenticated encryption with associated data (AEAD) [PR20]. A combiner is a method to connect instantiations of a primitive, here AEAD, such that this combination remains secure as long as only one of its component instances is.

[PR20] Bertram Poettering and **Paul Rösler**. Combiners for AEAD. IACR Transactions on Symmetric Cryptology, 2020 (1)
The majority of formal results and textual descriptions in this work were contributed by the author of this thesis.

Co-Author As a co-author, we also contributed to an analysis of attacks against deterministic signature schemes [PSS+]. Our contribution therein was the contextualization and abstraction of the attack strategy with regards to underlying essential conditions that undermine the Fiat-Shamir transform [FS87]. Furthermore, the results of

the author's Bachelor's thesis [Rös15] were core contents of an article that analyzes security issues in the deployment of encrypted cloud infrastructures [GMR+16]. Inspired by this work, an analysis of weaknesses in Microsoft's document protection mechanisms [GMRS16] has been co-authored by the author of this thesis.

[PSS+] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and **Paul Rösler**. Attacking deterministic signature schemes using fault attacks. In IEEE European Symposium on Security and Privacy – EuroS&P 2018 (authors ordered by their contributions)
The abstraction and contextualization of the attack in this work were contributed by the author of this thesis.

[GMR+16] Martin Grothe, Christian Mainka, **Paul Rösler**, Johanna Jupke, Jan Kaiser, and Jörg Schwenk. Your cloud in my company: Modern rights management services revisited. In International Conference on Availability, Reliability and Security – ARES 2016 (authors ordered by their contributions)
The attack against Tresorit as well as significant parts of textual descriptions in this work were contributed by the author of this thesis.

[Rös15] **Paul Rösler**. Architektur- und Sicherheitsanalyse von Tresorit und Tresorit DRM. Bachelor's thesis, Ruhr University Bochum, 2015

[GMRS16] Martin Grothe, Christian Mainka, **Paul Rösler**, and Jörg Schwenk. How to break Microsoft rights management services. In USENIX Workshop on Offensive Technologies – WOOT 16 (authors ordered by their contributions)
Support for the attack validation as well as revisions of textual descriptions in this work were contributed by the author of this thesis.

1.5 Organization

We introduce our notation as well as necessary basic definitions in Chapter 2. The contents of the core chapters 3, 4, 5, and 6 have been sketched above. Notable work related to the results in these chapters are provided therein, respectively. We conclude this thesis and provide an outlook on extending future work and yet open questions posed by our results in Chapter 7.

2 Preliminaries

We here introduce our modeling approaches, and the notation and security definitions for all basic building blocks used in this entire work.

2.1 Cryptographic Modeling

In order to formally analyze the security of cryptographic protocols, we use abstractions of reality. We therefore make use of two different well established modeling techniques: computational game-based models and symbolic execution models.

We acknowledge the widespread perspective on modeling that "all models are wrong" [Box76] when comparing their abstractions with conditions in the real-world. Consequently, in addition to introducing these modeling techniques, we provide a short discussion on their practical meaning. However, since these general modeling approaches are not the focus of this work but only tools for analyzing security, we restrict their introduction and discussion to the necessary minimum and refer the interested reader to standard literature on computer sciences and cryptography for further details.

2.1.1 Computational Game-Based Models

A *computational game-based security model* defines security for a cryptographic protocol via a game played by an adversary who wins this game if it breaks the considered security property of this protocol. Such a game models the real-world interaction between practical attackers and their victims' protocol executions through oracles that the adversary can query during the game. With one type of oracles the

adversary can control (the schedule and public inputs of) algorithm invocations of victims to let them execute the protocol at its will. With another type of oracles the adversary can access the victims' secrets that they use in these invocations. To capture successful adversarial attacks in the game's winning condition, further oracles can be introduced that embed challenges for the adversary or accept respective solutions from it. We emphasize that this type of modeling does not further specify the adversarial behavior during the game—especially not the adversary's internal computations.

Almost all cryptographic protocols can be broken by computationally unrestricted attackers (except for, e.g., the one-time pad) and all practical attackers are indeed restricted in their computational power. Hence, a *computational game-based security model* defines the advantage of any adversary in winning such a game under consideration of respectively needed run-time and memory consumption. (Thereby an adversary that needs *unreasonably* many resources for its attack is declared unsuccessful.)

It has been a tradition in the cryptographic literature to define security according to a *security parameter*. From this parameter both the asymptotic maximum of computational resources used by a realistic adversary (polynomial in the security parameter) and the asymptotic maximum of its advantage in breaking the security property (negligible in the security parameter) are derived under which a protocol is declared *secure* accordingly. Modern cryptographic literature instead provides concrete advantage terms that clearly point out the relation between breaking a protocol and reducing this to breaks of its component building blocks under the resources needed for this reduction. While the concept of *security* is the same for both approaches, the concrete advantage term more precisely specifies the quality of a reduction and thereby rather states the conditions under which the practical deployment of a protocol is secure (e.g., which key lengths are appropriate and necessary). We follow this latter approach in this work.

Formal Notation of Games Security games, that we denote with **Game**, invoke probabilistic adversaries via instruction 'Invoke'. These adversaries are denoted by calligraphic letters (usually $\mathcal{A}$ or $\mathcal{B}$). Adversaries have access to the game's interface, which is defined by oracles that are denoted by the term **Oracle**. Games are terminated via instructions 'Stop with x' (meaning that x is returned by the game) or 'Reward b' (meaning that the game terminates and returns 1 if the Boolean variable b has the value True). The two instructions are used for appraising the actions of the adversary: Intuitively, if the adversary behaves such that a required condition is violated then the adversary definitely 'loses' the game, and if it behaves such that a rewarded condition is met then it definitely 'wins'. We write $\Pr[G \Rightarrow 1]$ for the probability that game G terminates with return value 1. In procedures that we denote by **Proc** and in oracles, we use the shortcut notion 'Require x'. Depending on the procedure's or oracle's number of return values n, that means 'If $x = \mathtt{F}$, then return $\perp^n$', where $\mathtt{F}$ is the Boolean constant False and $\perp$ is a special abortion symbol.

We note that there are different established ways to define security games related to key exchange. Some works give very compact definitions (in [BSJ+17] a ratcheting security notion is compressed, without losing detail, into a single figure), while other works specify game families, parameterized for instance with separate freshnesh predicates (in [CCD+17], security notions for ratcheting are divided into the game description and a description of the freshness predicate). We follow the former approach and give a discussion on the modeling of ratcheting in Section 3.13.

2.1.2 Symbolic Execution Models

In Chapter 5 we use an entirely different approach to model adversaries, cryptographic protocols, and the interaction between them: A *symbolic execution model* treats all values in an environment as abstract symbols that have no bit representation, and hence neither an algebraic structure. For computations with these symbols, all algorithms in the environment, including those of the protocol and the ad-

versary itself, are defined to follow predefined derivation rules. These derivation rules model the computational power of available (cryptographic) building blocks. A cryptographic protocol is declared *secure* if no symbolic adversarial execution can reach a state in its symbolic execution of the analyzed protocol that is defined as a break of the protocol's security properties. Adversaries in symbolic execution models, in contrast to computational models, are unbounded in their run-time and memory consumption. Nevertheless, the restriction of available computations to fixed derivation rules is a heavy (unrealistic) idealization. Thereby the meaning of a proven statement in a symbolic model heavily depends on the (computational) power provided by these derivation rules.

Formal Notation of Symbolic Execution Models A symbolic execution model mainly consists of the definition of types of symbols, grammar rules that describe the relation between these types, and derivation rules that describe possible transitions between the symbols. We describe grammar rules as follows: For three types of symbols X, Y, and Z in a grammar, $X \mapsto Y|Z$ denotes that symbols of type X can be parsed as symbols of type Y or type Z. A type that cannot be parsed further is called *terminal type*. Using these grammar rules, we define derivation rules that describe how symbols can be derived from sets of (other) symbols. For a symbol m and set of symbols $\boldsymbol{M}$, $\boldsymbol{M} \vdash m$ means that m can be derived from the symbols in set $\boldsymbol{M}$ by using the grammar and derivation rules that we specify in our symbolic model.

2.2 Notation

Our general notation for the description and use of algorithms, sets, intervals, lists, etc. is defined as follows. If A is a (deterministic or probabilistic) algorithm we write $A(x)$ for an invocation of A on input x. If A is probabilistic, we write $A(x) \Rightarrow y$ for the event that an invocation results in value y being the output. We further write

$[A(x)] := \{y : \Pr[A(x) \Rightarrow y] > 0\}$ for the effective range of $A(x)$.

If $a \leq b$ are integers, we write $[a\,..\,b]$ for the set $\{a, \ldots, b\}$ and we write $[\,a, ...\,]$ for the set $\{x \in \mathbb{N} : a \leq x\}$. We refer to intervals and their boundaries (smallest and largest elements) as follows: For an interval $I = [a\,..\,b]$ we write $I^{\lhd}$ for a and $I^{\rhd}$ for b. We denote the Boolean constants True and False with `T` and `F`, respectively. We use Iverson brackets to convert Boolean values into bit values: $[\mathtt{T}] = 1$ and $[\mathtt{F}] = 0$. To compactly write if-then-else expressions we use the ternary operator known from the C programming language: If C is a Boolean condition and e_1, e_2 are arbitrary expressions, the composed expression "C `?` e_1 `:` e_2" evaluates to e_1 if $C = \mathtt{T}$ and to e_2 if $C = \mathtt{F}$.

Symbol 'ϵ' denotes an empty string. When we refer to a *list* or *sequence* we mean a (row) vector that can hold arbitrary elements, where we abuse the notation of empty string 'ϵ' to denote the empty list. Lists that hold precisely one element are notationally identified with the element itself. By $\mathcal{X}^*$, we denote the set of all lists of arbitrary size whose elements belong to $\mathcal{X}$. With $\mathcal{P}(\mathcal{X})$ we denote the power set of $\mathcal{X}$ such that $\mathcal{P}(\mathcal{X})$ is the set of all subsets of $\mathcal{X}$. If an element or a list $x \in \mathcal{X}^*$ is appended to list L then we denote this by $L \leftarrow L \,\|\, x$ (or simply $L \xleftarrow{\shortparallel} x$). Thus, '$\|$' denotes a special concatenation symbol that is not an element of any of the explicitly defined sets. We define relations prefix-or-equal $\preceq$ and strictly-prefix $\prec$ over two lists. For instance, for lists $L, L_0 = L \,\|\, x, L_1 = L \,\|\, y$ where $x, y \in \mathcal{X}, x \neq y$ we have that $L \preceq L, L \not\prec L, L \prec L_0, L \prec L_1, L_0 \not\preceq L_1, L_1 \not\preceq L_0$ meaning that L is a prefix of L_0 and L_1 but neither of L_0, L_1 is a prefix of the other. Note that if the elements held by two lists are strings (over some alphabet) then the concatenation of the lists does not result in the strings being concatenated; in particular, `"ab"` $\|$ `"c"` $\neq$ `"abc"`. (We do not use string concatenation in this work, so ambiguities are naturally avoided.) We denote the cardinality of a set $\mathcal{X}$ or the length of a string s with symbols $|\mathcal{X}|$ and $|s|$.

Program code. We describe algorithms and security experiments using (pseudo-)code. In such code we distinguish the following operators for assigning values to variables: We use symbol '$\leftarrow$' when

the assigned value results from a constant expression (including the output of a deterministic algorithm), and we write '$\leftarrow_{\$}$' when the value is either sampled uniformly at random from a finite set or is the output of a probabilistic algorithm. For such a probabilistic algorithm Y, $x \leftarrow \mathrm{Y}(y;r)$ denotes the deterministic evaluation of Y on y with output x where the evaluation's randomness is fixed to r.

If we assign a value that is a tuple but we are actually not interested in some of its components, we use symbol '_' to mark positions that shall be ignored. For instance, $(\_, b, \_) \leftarrow (A, B, C)$ is equivalent to $b \leftarrow B$. If X, Y are sets we write $X \xleftarrow{\cup} Y$ shorthand for $X \leftarrow X \cup Y$, and if L_1, L_2 are lists we write $L_1 \xleftarrow{\shortparallel} L_2$ shorthand for $L_1 \leftarrow L_1 \,\|\, L_2$. We use bracket notation to denote associative arrays (a data structure that implements a dictionary). Associative arrays can be indexed with elements from arbitrary sets. For instance, for an associative array A the instruction $A[7] \leftarrow 3$ assigns value 3 to index 7, and the expression $A[\mathtt{abc}] = 5$ tests whether the value at index `abc` is equal to 5. We write $A[\cdot] \leftarrow x$ to initialize the associative array A by assigning the default value x to all possible indices. For an integer a we write $A[..., a] \leftarrow x$ as a shortcut for 'For all $a' \leq a$: $A[a'] \leftarrow x$'.

Scheme specifications. We also describe the algorithms of cryptographic schemes using program code. Some algorithms may abort or fail, indicating this by outputting the special symbol $\perp$. This is implicitly assumed to happen whenever an encoded data structure provided by the adversary is to be parsed into components but the encoding turns out to be invalid (thus $\perp$ is not an element of explicitly defined sets).

2.3 Cryptographic Building Blocks

In the following we define computational game-based security for standard cryptographic primitives used as building blocks in this thesis. For clarity in our proofs, we consider multi-instance notions of security below. These notions are equivalent to the respective standard notions with only one instance. However, the corresponding trivial

reduction loses a factor of n, where n is the total number of instances: The instance which the adversary successfully attacks is guessed in the reduction, and the $n-1$ remaining instances are simulated with knowledge of the corresponding key. Beyond clarity, using multi-instance notions reduces unnecessary tightness losses in our reductions that are induced by model artifacts.

2.3.1 (Dual) Pseudo-Random Function

A *pseudo-random function* (PRF) for an associated-data space $\mathcal{AD}$ and a samplable key space $\mathcal{K}$ is a function $\mathsf{PR} = \mathrm{prf}$ that takes a key $k \in \mathcal{K}$ and an associated-data string $ad \in \mathcal{AD}$, and outputs another key $k' \in \mathcal{K}$. A *dual* PRF for a samplable key space $\mathcal{K}$ is a function $\mathsf{PR} = \mathrm{dprf}$ that takes two keys $k_1, k_2 \in \mathcal{K}$ and outputs another key $k' \in \mathcal{K}$. Shortcut notations are thus

$$\mathcal{K} \times \mathcal{AD} \to \mathrm{prf} \to \mathcal{K} \qquad \mathcal{K} \times \mathcal{K} \to \mathrm{dprf} \to \mathcal{K} \ .$$

We only define computational game-based security for standard PRFs here as dual PRFs are only considered symbolically in Chapter 5.

For security of (standard) PRFs we formalize a multi-instance variant of indistinguishability of output keys of a keyed scheme PR from output (keys) of a random function which we denote by game $\mathrm{KIND}^b_{\mathsf{PR}}$ from Figure 2.1. In this game, the adversary can choose associated-data inputs to either evaluate the actual PRF on one out of multiple keys, or evaluate one out of multiple random functions accordingly. The adversary is also allowed to create new instances, or to expose them, meaning to learn their keys. The advantage of an adversary $\mathcal{A}$ in distinguishing evaluations of the PRF from evaluations of random functions in game $\mathrm{KIND}^b_{\mathsf{PR}}$ is defined as $\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{PR}}(\mathcal{A}) := \Pr[\mathrm{KIND}^0_{\mathsf{PR}}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathrm{KIND}^1_{\mathsf{PR}}(\mathcal{A}) \Rightarrow 1]$. Intuitively, a PRF is secure if all practical adversaries have a negligible advantage.

A secure dual PRF additionally achieves indistinguishability of output keys in case at most one of the two input keys is exposed. For simplicity (in our proof), we only consider *symmetric* dual PRFs [BL15],

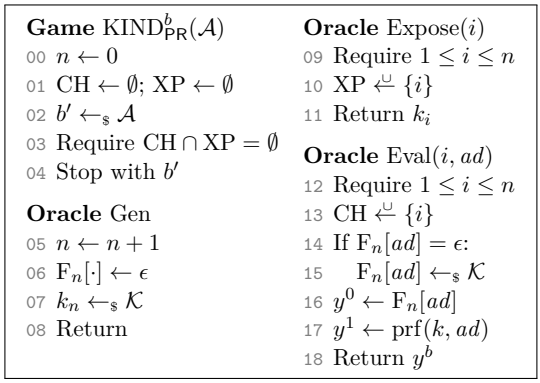

Figure 2.1: Security experiment KIND, modeling the security of a pseudo-random function in a multi-instance setting. Variable n indicates the number of established instances, associated array F_i simulates a random function for each instance i, and sets CH and XP keep track of the instances that are challenged due to an evaluation and exposed, respectively.

fulfilling the property that $\mathrm{dprf}(k_1, k_2) = \mathrm{dprf}(k_2, k_1) = k'$ for all $k_1, k_2 \in \mathcal{K}$.

2.3.2 Message Authentication Codes

A *message authentication code* (MAC) for a message space $\mathcal{M}$ is a pair $\mathsf{M} = (\mathrm{tag}, \mathrm{vfy}_{\mathsf{M}})$ of algorithms together with a samplable key space $\mathcal{K}$ and a tag space $\mathcal{T}$. The tag-generation algorithm tag takes a key $k \in \mathcal{K}$ and a message $m \in \mathcal{M}$, and outputs a tag $\tau \in \mathcal{T}$. The deterministic tag-verification algorithm $\mathrm{vfy}_{\mathsf{M}}$ takes a key $k \in \mathcal{K}$, a message $m \in \mathcal{M}$, and a tag $\tau \in \mathcal{T}$, and outputs a Boolean value: either $\mathtt{T}$ (for accept) or $\mathtt{F}$ (for reject). Shortcut notations for tag generation and verification are thus

$$\mathcal{K} \times \mathcal{M} \to \mathrm{tag} \to \mathcal{T} \qquad \mathcal{K} \times \mathcal{M} \times \mathcal{T} \to \mathrm{vfy}_{\mathsf{M}} \to \{\mathtt{T}, \mathtt{F}\} \ .$$

For correctness we require that for all $k \in \mathcal{K}$ and $m \in \mathcal{M}$ and $\tau \in [\mathrm{tag}(k, m)]$ we have $\mathrm{vfy}_{\mathsf{M}}(k, m, \tau) = \mathtt{T}$.

As a security property for MACs we formalize a multi-instance version of one-time strong unforgeability. In this notion the adversary

has to produce, for a message of its choice, a fresh but valid tag (for any out of a set of independent instances, each initialized with a uniformly picked key). The adversary can create new instances or to expose them, and is supported by tag generation and verification oracles, where the former can only be queried once per instance. The details of this notion are in game SUF in Figure 2.2. For a MAC M, we associate with any adversary $\mathcal{A}$ its strong unforgeability advantage $\mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{A}) := \Pr[\mathrm{SUF}_{\mathsf{M}}(\mathcal{A}) \Rightarrow 1]$. Intuitively, a MAC is secure if all practical adversaries have a negligible advantage.

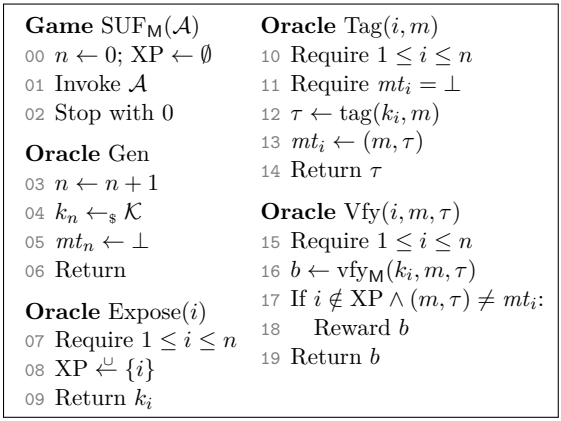

Figure 2.2: Security experiment SUF, modeling the one-time strong unforgeability of a MAC in a multi-instance setting. Variable n indicates the number of established instances, set XP keeps track of the instances that are exposed, and for each instance i variable mt_i keeps track of the message and corresponding tag that are processed in Tag queries.

2.3.3 Signature Schemes

A *signature scheme* for a message space $\mathcal{M}$ is a triple $\mathsf{S} = (\mathrm{gen}_{\mathsf{S}}, \mathrm{sgn}, \mathrm{vfy}_{\mathsf{S}})$ of algorithms together with a signer key-space $\mathcal{SK}$, a verifier key-space $\mathcal{VK}$, and a signature space Σ. The randomized key-generation algorithm $\mathrm{gen}_{\mathsf{S}}$ outputs a signer key $sgk \in \mathcal{SK}$ and a verifier key $vfk \in \mathcal{VK}$. The signing algorithm sgn may be randomized and takes a

signer key $sgk \in \mathcal{SK}$ and a message $m \in \mathcal{M}$, and outputs a signature $\sigma \in \Sigma$. The deterministic verification algorithm $\mathrm{vfy}_{\mathsf{S}}$ takes a verifier key $vfk \in \mathcal{VK}$, a message $m \in \mathcal{M}$, and a (candidate) signature $\sigma \in \Sigma$, and outputs a bit $b \in \{\mathtt{T}, \mathtt{F}\}$, indicating acceptance and rejection, respectively. Shortcut notations for the three algorithms are thus

$$\mathrm{gen}_{\mathsf{S}} \to_{\$} \mathcal{SK} \times \mathcal{VK} \qquad \mathcal{SK} \times \mathcal{M} \to \mathrm{sgn} \to_{\$} \Sigma$$

$$\mathcal{VK} \times \mathcal{M} \times \Sigma \to \mathrm{vfy}_{\mathsf{S}} \to \{\mathtt{T}, \mathtt{F}\} \ .$$

For correctness we require that for all $(sgk, vfk) \in [\mathrm{gen}_{\mathsf{S}}]$ and $m \in \mathcal{M}$ and $\sigma \in [\mathrm{sgn}(sgk, m)]$ we have $\mathtt{T} = \mathrm{vfy}_{\mathsf{S}}(vfk, m, \sigma)$.

We formalize a multi-instance security notion of one-time strong unforgeability for signatures. Concretely, the adversary controls the messages processed by the signers, it sees the resulting signatures, and its goal is to make a verifier accept a message-signature pair that was not processed by the respective signer. Again, the adversary can create new instances or expose their singing keys. The details of this notion are in game SUF in Figure 2.3. For a signature scheme S, we associate with any adversary $\mathcal{A}$ its strong unforgeability advantage $\mathrm{Adv}^{\mathrm{suf}}_{\mathsf{S}}(\mathcal{A}) := \Pr[\mathrm{SUF}_{\mathsf{S}}(\mathcal{A}) \Rightarrow 1]$. Intuitively, a signature scheme is secure if all practical adversaries have a negligible advantage.

2.3.4 Key Encapsulation Mechanisms

We consider a type of key encapsulation mechanism where key pairs are generated by first randomly sampling the secret key and then deterministically deriving the public key from it. While this syntax is non-standard, note that it can be assumed without loss of generality: One can always understand the coins used for (randomized) key generation of a classic key encapsulation mechanism as the secret key in our sense.

A *key encapsulation mechanism* (KEM) for a finite symmetric-key space $\mathcal{K}$ is a triple $\mathsf{K} = (\mathrm{gen}_{\mathsf{K}}, \mathrm{enc}, \mathrm{dec})$ of algorithms together with a samplable secret-key space $\mathcal{SK}$, a public-key space $\mathcal{PK}$, and a ciphertext space $\mathcal{C}$. In its regular form the public-key generation algorithm $\mathrm{gen}_{\mathsf{K}}$ is deterministic, takes a secret key $sk \in \mathcal{SK}$, and outputs

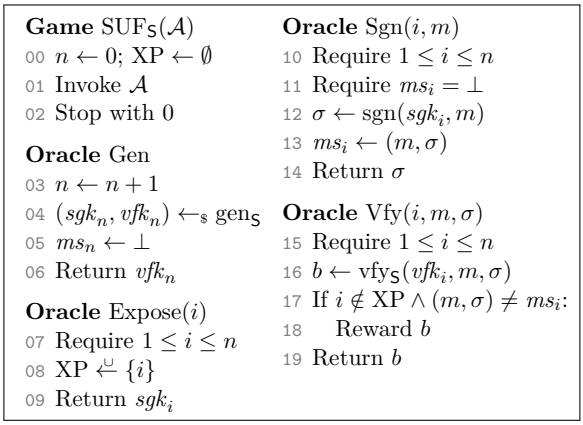

Figure 2.3: Security experiment SUF, modeling the one-time strong unforgeability of a signature scheme in a multi-instance setting. Variable ms records the message-signature combination processed by the signer.

a public key $pk \in \mathcal{PK}$. We also use a shorthand form, writing gen_{K} for the randomized procedure of first picking $sk \leftarrow_{\$} \mathcal{SK}$, then deriving $pk \leftarrow \text{gen}_{\mathsf{K}}(sk)$, and finally outputting the pair (sk, pk). Two shortcut notations for key generation are thus

$$\mathcal{SK} \rightarrow \text{gen}_{\mathsf{K}} \rightarrow \mathcal{PK} \qquad \text{gen}_{\mathsf{K}} \rightarrow_{\$} \mathcal{SK} \times \mathcal{PK} \ .$$

The randomized encapsulation algorithm enc takes a public key $pk \in \mathcal{PK}$ and outputs a symmetric key $k \in \mathcal{K}$ and a ciphertext $c \in \mathcal{C}$, and the deterministic decapsulation algorithm dec takes a secret key $sk \in \mathcal{SK}$ and a ciphertext $c \in \mathcal{C}$, and outputs either a symmetric key $k \in \mathcal{K}$ or the special symbol $\perp \notin \mathcal{K}$ to indicate rejection. Shortcut notations for encapsulation and decapsulation are thus

$$\mathcal{PK} \rightarrow \text{enc} \rightarrow_{\$} \mathcal{K} \times \mathcal{C} \qquad \mathcal{SK} \times \mathcal{C} \rightarrow \text{dec} \rightarrow \mathcal{K} \cup \{\perp\} \ .$$

For correctness we require that for all $(sk, pk) \in [\text{gen}_{\mathsf{K}}]$ and $(k, c) \in [\text{enc}(pk)]$ we have $\text{dec}(sk, c) = k$.

For security of KEMs we formalize a multi-receiver/multi-challenge version of one-way security—as opposed to strictly stronger key indistinguishability—which is sufficient for our purposes. In this notion,

the adversary obtains challenge ciphertexts and has to recover any of the encapsulated keys. The adversary is supported by a key-checking oracle that, for a provided pair of ciphertext and (candidate) symmetric key, tells whether the ciphertext decapsulates to the indicated key. The adversary is also allowed to establish new receivers, or to expose them, meaning to learn their secret keys. The details of this notion are in game OW in Figure 2.4. For a KEM K, we associate with any adversary $\mathcal{A}$ its one-way advantage $\mathrm{Adv}^{\mathrm{ow}}_{\mathsf{K}}(\mathcal{A}) := \Pr[\mathrm{OW}_{\mathsf{K}}(\mathcal{A}) \Rightarrow 1]$. Intuitively, a KEM is secure if all practical adversaries have a negligible advantage.

Game $\mathrm{OW}_{\mathsf{K}}(\mathcal{A})$
00 $n \leftarrow 0$; $\mathrm{XP} \leftarrow \emptyset$
01 Invoke $\mathcal{A}$
02 Stop with 0

Oracle Gen
03 $n \leftarrow n + 1$
04 $(sk_n, pk_n) \leftarrow_{\$} \mathrm{gen}_{\mathsf{K}}$
05 $\mathrm{CK}_n[\cdot] \leftarrow \perp$
06 Return pk_n

Oracle $\mathrm{Enc}(i)$
07 Require $1 \le i \le n$
08 $(k, c) \leftarrow_{\$} \mathrm{enc}(pk_i)$
09 $\mathrm{CK}_i[c] \leftarrow k$
10 Return c

Oracle $\mathrm{Solve}(i, c, k)$
11 Require $1 \le i \le n$
12 Require $i \notin \mathrm{XP}$
13 Require $\mathrm{CK}_i[c] \neq \perp$
14 Reward $k = \mathrm{CK}_i[c]$
15 Return

Oracle $\mathrm{Check}(i, c, k)$
16 Require $1 \le i \le n$
17 $k' \leftarrow \mathrm{dec}(sk_i, c)$
18 Return $[k' = k]$

Oracle $\mathrm{Expose}(i)$
19 Require $1 \le i \le n$
20 $\mathrm{XP} \xleftarrow{\cup} \{i\}$
21 Return sk_i

Figure 2.4: Security experiment OW, modeling the one-way security of a KEM in a multi-receiver/multi-challenge setting. Variable n indicates the number of established receivers, set XP keeps track of the receivers that are exposed, and for each receiver i the associative array CK_i keeps track of the ciphertexts and symmetric keys that are processed in Enc queries.

Our variant of one-wayness is equivalent to the standard notion with only one receiver, one challenge encapsulation, and no exposure. However, the corresponding reduction loses a factor of nm,[1] where n is the total number of receivers and m is the total number of chal-

[1] For all KEM applications in this thesis we actually have $m = 1$, i.e., the security loss is effectively only by a factor of n.

lenge encapsulations per receiver: The receiver for which the adversary successfully recovers the symmetric key is guessed, and the $n-1$ remaining receivers are simulated with knowledge of their secret key; further, the later-broken encapsulation query of the identified user is guessed, and the remaining encapsulation queries are simulated using the regular encapsulation algorithm.

2.3.5 Further Building Blocks

Cryptographic building blocks that we only consider informally in this thesis or exclusively in our symbolic model in Chapter 5 are hierarchical identity-based KEM and broadcast encryption. Below we define their syntax formally but only provide an intuition for their security guarantees. For formal security definitions we refer the reader to the respective literature (e.g., [GS02, FN94]).

Hierarchical Identity-Based Encryption *Hierarchical identity-based key encapsulation mechanism* (HIBE)[2] is a variant of KEM where the encapsulation algorithm takes, in addition to the public key, an identity vector. From the according secret key further identity-specific secret keys can be delegated such that an identity vector as input to the encapsulation specifies, which delegated identity-specific secret keys can be used for *successful* decapsulation.

Formally, an HIBE scheme for a finite symmetric-key space $\mathcal{K}$ and a finite identity space $\mathcal{ID}$ is a quadruple $\mathsf{HK} = (\mathrm{gen}_{\mathsf{HK}}, \mathrm{enc}, \mathrm{dec}, \mathrm{del})$ of algorithms together with a samplable secret-key space $\mathcal{SK}$, a public-key space $\mathcal{PK}$, and a ciphertext space $\mathcal{C}$. Algorithms $\mathrm{gen}_{\mathsf{HK}}$ and dec conform their KEM equivalents. The randomized encapsulation algorithm enc takes a public key $pk \in \mathcal{PK}$ and an identity vector $id \in \mathcal{ID}^*$, and outputs a symmetric key $k \in \mathcal{K}$ and a ciphertext $c \in \mathcal{C}$, and the delegation algorithm del takes a secret key $sk \in \mathcal{SK}$ and an identity string $id \in \mathcal{ID}$, and outputs another secret key $sk \in \mathcal{SK}$. Shortcut

[2] For simplicity, we avoid the acronym HIB-KEM here.

notations for encapsulation and delegation are thus

$$\mathcal{PK} \times \mathcal{ID}^* \to \text{enc} \to_\$ \mathcal{K} \times \mathcal{C} \qquad \mathcal{SK} \times \mathcal{ID} \to \text{del} \to \mathcal{SK} \ .$$

For correctness we require that for all $(sk_0, pk) \in [\text{gen}_{\mathsf{HK}}]$ and $id \in \mathcal{ID}^*$, with $id = id_1 \,\|\, \ldots \,\|\, id_l$ such that $\forall i \in [l]$ $id_i \in \mathcal{ID}$ and $sk_i = \text{del}(sk_{i-1}, id_i)$, and $(k, c) \in [\text{enc}(pk, id)]$ we have $\text{dec}(sk_l, c) = k$.

A secure HIBE intuitively guarantees that a symmetric key, encapsulated to a public key pk and an identity vector $id \in \mathcal{ID}^*$, cannot be decapsulated by a (practical) adversary even with access to any secret keys, derived via delegations from pk's secret key sk under an identity vector $id' \in \mathcal{ID}^*$ such that id' is not a prefix of id.

Broadcast Encryption *Broadcast encryption* (BE) is a variant of public key encryption where the encryption algorithm takes, in addition to a message and a (main) public key, a subset of the natural numbers. This subset refers to users—each of them referenced by a unique contained number—who are excluded from being able to decrypt the encrypted message with their secret key that they registered from the according main secret key.

Formally, a BE scheme for a message space $\mathcal{M}$ is a quadruple $\mathsf{BE} = (\text{gen}_{\mathsf{BE}}, \text{enc}_{\mathsf{BE}}, \text{dec}, \text{reg})$ of algorithms together with a samplable main secret-key space $\mathcal{MSK}$, a (registered) secret-key space $\mathcal{SK}$, a (main) public-key space $\mathcal{MPK}$, and a ciphertext space $\mathcal{C}$. Algorithms gen_{BE} and dec conform their KEM equivalents in principle, except that algorithm gen_{BE} outputs a *main* key pair (instead of a standard key pair) and algorithm dec generically outputs a message $m \in \mathcal{M}$ (instead of a key as usual for public key encryption). Shortcut notations for key pair generation and decryption are thus

$$\mathcal{MSK} \to \text{gen}_{\mathsf{BE}} \to \mathcal{MPK} \quad \text{gen}_{\mathsf{BE}} \to_\$ \mathcal{MSK} \times \mathcal{MPK}$$

$$\mathcal{SK} \times \mathcal{C} \to \text{dec} \to \mathcal{M} \cup \{\perp\}.$$

The randomized encryption algorithm enc_{BE} takes a (main) public key $pk \in \mathcal{MPK}$ a finite set $\boldsymbol{RM} \subset \mathbb{N}$, and a message $m \in \mathcal{M}$, and

outputs a ciphertext $c \in \mathcal{C}$, and the randomized registration algorithm reg takes a main secret key $msk \in \mathcal{MSK}$ and a natural number $u \in \mathbb{N}$, and outputs a registered secret key $sk \in \mathcal{SK}$. Shortcut notations for encryption and registration are thus

$$\mathcal{MPK} \times \mathcal{P}(\mathbb{N}) \times \mathcal{M} \to \mathrm{enc}_{\mathsf{BE}} \to_{\$} \mathcal{C} \qquad \mathcal{MSK} \times \mathbb{N} \to \mathrm{reg} \to \mathcal{SK} \ .$$

For correctness we require that for all $(msk, mpk) \in [\mathrm{gen}_{\mathsf{BE}}]$ and $u \in \mathbb{N}$ and $sk \in [\mathrm{reg}(msk, u)]$ and $\boldsymbol{RM} \subset \mathbb{N} \setminus \{u\}$ and $c \in [\mathrm{enc}_{\mathsf{BE}}(mpk, \boldsymbol{RM}, m)]$ we have $\mathrm{dec}(sk, c) = m$.

A secure BE scheme intuitively guarantees that a message, encrypted to a (main) public key mpk with a set of removed users $\boldsymbol{RM}$, cannot be decrypted by a (practical) adversary even with access to any secret keys, registered under mpk's main secret key msk for numbers $u \in \boldsymbol{RM}$.

We note that our (correctness) notion of BE requires that from each main secret key an infinite number of secret keys can be registered. Although this is neither standard nor practical, this notion of BE used in the symbolic model of Chapter 5 strengthens our results (with respect to practice).

2.3.6 Random Oracle Model

For allowing simple, comprehensible, and clear proofs of security in this work, in some of our reduction proofs we make use of the random oracle model that idealizes (security) guarantees of hash functions.

In the *random oracle model* a hash function H with input of arbitrary length bit strings from space $\{0,1\}^*$ and output of fixed length bit strings from space $\{0,1\}^l$ is modeled as a random function from the set of all functions over these input and output spaces. Shortcut notion for the modeled hash function is thus

$$\{0,1\}^* \to \mathrm{H} \to \{0,1\}^l \ .$$

Adversaries, attacking constructions in the random oracle model, only have oracle-access to function H, meaning that for input $x \in \{0,1\}^*$

they can obtain output $y \in \{0,1\}^l$ only by querying the model on x where $y = \mathrm{H}(x)$. In particular, adversaries are not able to directly access the function description of H in order to evaluate H on input x themselves.

When being deployed, schemes proven secure in the random oracle model are implemented with a hash function that is assumed to behave like a random function (i.e., whose outputs are indistinguishable from random outputs for practical adversaries). It has to be noted that there exist example schemes, provably secure in the random oracle model, that become entirely insecure when the modeled hash function is instantiated by any practically implementable (hash) function [CGH98b, BBP04, MRH04]. These schemes are, however, contrived and artificial and therefore do not invalidate the meaning of proofs in the random oracle for real-world implementations of respectively analyzed practical schemes per se. The random oracle model particularly simplifies security analyses of practical cryptographic protocols, and as of yet there exists no attack against a practical scheme that is caused by the gap between the random oracle model and a respective instantiation with a real hash function. Hence, the use of the random oracle model is a widespread approach in the cryptographic literature.

As a result, we agree with Shai Halevi's conclusion [CGH98a, 6.3] that proofs in the random oracle model appear to be useful since it is unclear whether the mentioned counterexamples apply to entirely different uses of random oracles. Furthermore, we note that many other unrealistic idealizing assumptions in common security analyses represent significantly more serious gaps between proved statements and actual security guarantees of practical deployments. To mention two of these gaps, considered and closed in our work, secrets of victims are in many security models inaccessible for adversaries during the entire lifetime of their protocol execution, and random coins of probabilistic algorithm invocations are in many security models sampled from a uniform distribution. Neither of both can be guaranteed in practice. Another significant idealization are symbolic execution models that were already introduced and discussed in Section 2.1.2.

Hence, the ultimate goal of cryptographic analyses is to aim for security proofs that hold in the standard model (i.e., not in the random oracle model) under consideration of exposure of secrets, deviations in randomness distributions, subversion of implementations, leakage of execution time, etc. Nevertheless, security proofs that make (partially or even entirely) unrealistic idealizations can be initial steps of cryptographic analyses and regularly support the understanding of the considered constructions.

3

Optimally Secure Ratcheting in Two-Party Settings

Contents

3.1 Introduction 41
3.2 Key-updatable Key Encapsulation Mechanisms 47
3.3 Unidirectionally ratcheted key exchange (URKE) . . . 51
3.4 Constructing URKE 57
3.5 Sesquidirectionally ratcheted key exchange (SRKE) . . 60
3.6 Constructing SRKE 66
3.7 Rationales for SRKE Design 72
3.8 Bidirectionally ratcheted key exchange (BRKE) 78
3.9 Constructing BRKE 80
3.10 Proof of URKE 86
3.11 Proof of SRKE 95
3.12 Proof of BRKE 113
3.13 Modeling ratcheted key exchange 117

Ratcheted key exchange (RKE) is a cryptographic technique used in almost all modern instant messaging systems (e.g., Signal and the WhatsApp messenger) for attaining strong security in the face of state exposure attacks. Abstractly, RKE continuously computes symmetric keys for two (or more) parties that they can use in higher level symmetric protocols (e.g., a secure channel). During the computation of these symmetric keys, each participant updates their used local state such that exposures thereof only have temporary effect on the keys' secrecy.

RKE initially received academic attention in the recent works of Cohn-Gordon et al. [CCD+17] and Bellare et al. [BSJ+17]. While the former is analytical in the sense that it aims primarily at assessing the security that one particular protocol *does* achieve (which might be weaker than the notion that it *should* achieve), the authors of the latter develop and instantiate a notion of security from scratch, independently of existing implementations. Unfortunately, however, their model is quite restricted, e.g. for considering only unidirectional communication and the exposure of only one of the two communication participants.

In this chapter, of which parts previously were published as an extended abstract in the proceedings of CRYPTO 2018 [PR18b], we resolve the limitations of prior work by developing alternative security definitions, for unidirectional RKE as well as for RKE where both participants contribute. We follow a purist approach, aiming at finding strong yet convincing notions that cover a realistic communication model with fully concurrent operation of both participants. We further propose secure instantiations (as the protocols analyzed or proposed by Cohn-Gordon et al. and Bellare et al. turn out to be weak in our models). While our scheme for the unidirectional case builds on a generic KEM as the main building block (differently to prior work that requires explicitly Diffie–Hellman), our schemes for bidirectional RKE require a stronger, HIBE-like component.

Contributions by the Author All formal work in this chapter except for the correctness and security games from figures 3.5, 3.6, 3.8, 3.9, 3.11, and 3.12 was contributed by the author of this thesis. Both authors of the original paper [PR18b] participated in writing the textual descriptions but the majority in this chapter was contributed by the author of this thesis, as well. The extended abstract [PR18b] that was published in the proceedings of CRYPTO 2018 only covers security definitions and constructions of unidirectional RKE and sesquidirectional RKE (from sections 3.3 and 3.5, and sections 3.4 and 3.6, respectively). This chapter additionally contains formal se-

curity proofs for these constructions (in sections 3.10 and 3.11) as well as a security definition of bidirectional RKE and an instantiation thereof that is also proven secure (in sections 3.8, 3.9, and 3.12). Furthermore, we here discuss our approach of defining security and compare it with classical models for authenticated key agreement (see Section 3.13).

3.1 Introduction

While the word ratcheting is sometimes associated with a set of techniques deployed with the aim of achieving certain (typically not formally defined) security goals, Bellare et al. recently pursued a different approach by proposing *ratcheted key exchange* (RKE) as a cryptographic primitive with clearly defined syntax, functionality, and security properties [BSJ+17]. This primitive establishes a sequence of symmetric session keys that allows for the construction of higher-level protocols, where instant messaging is just one example.[1] Building a messaging protocol on top of RKE—and thereby splitting the computation of symmetric keys and the protection of payload messages that uses these keys—offers clear advantages over using ad-hoc designs (as all messaging apps we are aware of do): the modularity allows for easier cryptanalysis, the substitution of constructions by alternatives, etc. We note, however, that the RKE formalization considered in [BSJ+17] is too limited to serve directly as a building block for secure messaging. In particular, the syntactical framework requires all communication to be unidirectional (in the Alice-to-Bob direction), and the security model counterintuitively assumes that exclusively Alice's local state can be exposed. (Unidirectional RKE is described in the top left part of Figure 3.1; note that the syntax in [BSJ+17] omits associated-data

[1] Note that RKE, despite its name, is a tool to be used in the 'symmetric phase' of a protocol that follows the preliminary key agreement. This key agreement initially provides two protocol participants a shared key from which they can derive corresponding local states (e.g., to initiate an RKE session). In [BSJ+17], and also in this chapter, the initial key agreement is abstracted away into a dedicated state initialization algorithm (or: protocol).

inputs.)

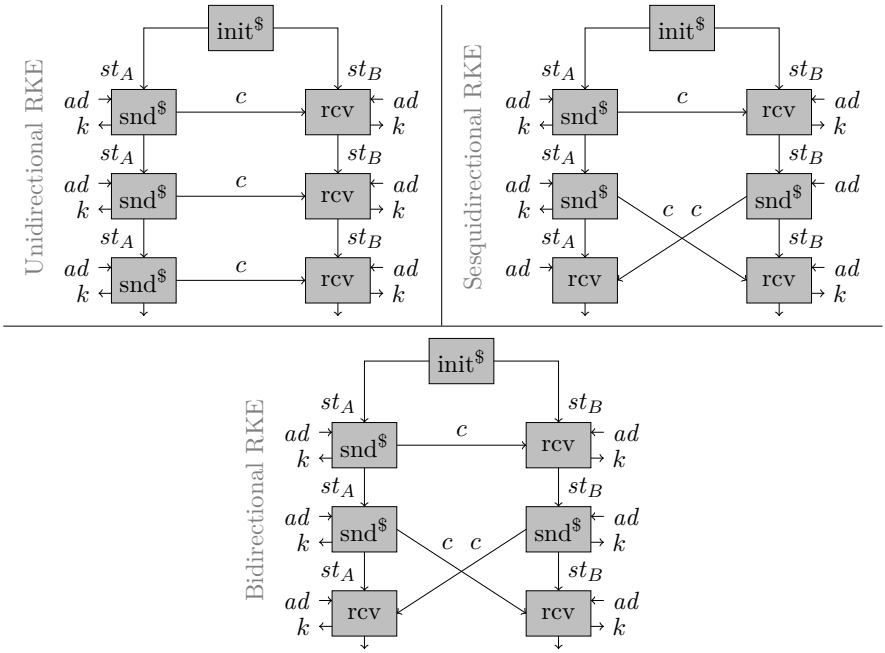

Figure 3.1: Concept of uni-, sesqui-, and bidirectional RKE: In unidirectional RKE Alice only sends and Bob only receives, and in bidirectional RKE both parties participate equally. Bob sending to Alice in sesquidirectional RKE only allows Bob to update his state and to share corresponding information with Alice without establishing new symmetric keys. '$' in the upper index of an algorithm indicates that it runs probabilistically and *ad* is associated data.

We give more details on the results of [BSJ+17]. In the proposed protocol, Alice's state has the form $st_A = (i, k.p, Y)$, where integer i counts her send operations, $k.p$ is a key for a PRF prf, and $Y = g^y$ is a public key of Bob. Bob's state has the form $st_B = (i, k.p, y)$. When Alice performs a send operation, she samples fresh randomness x, computes $\tau \leftarrow \mathrm{prf}(k.p, g^x)$ and $(k, k.p') \leftarrow \mathrm{H}(i, \tau, g^x, Y^x)$ where prf is a PRF and H is a random oracle, and outputs k as the established symmetric session key and $c = (g^x, \tau)$ as a ciphertext that is sent to Bob.

(Value τ serves as a message authentication code for g^x.) The next round's PRF key is $k.p'$, i.e., Alice's new state is $st_A = (i+1, k.p', Y)$. In this protocol, observe that prf and H together implement a 'hash chain' and lead to forward-secrecy, while the g^x, Y^x inputs to the random oracle can be seen as implementing one DHKE per transmission (where one exponent is static). Turning to the accordingly proposed RKE security model, while the corresponding game offers an oracle that allows adversaries to compromise Alice's state, there is no option for similarly exposing Bob. If the model had a corresponding oracle, the protocol would actually not be secure. Indeed, the following (fully passive) attack exploits that Alice 'encrypts' to always the same key Y of Bob: The adversary first reveals Alice's local state, learning $st_A = (i, k.p, Y)$; it then makes Alice invoke her send routine a couple of times and delivers the respective ciphertexts to Bob's receive routine in unmodified form; in the final step the adversary exposes Bob and recovers his past symmetric session keys using the revealed constant exponent y from state st_B. Note that in a pure RKE sense these session keys could (and should) remain unknown to the adversary: Alice should have recovered from the state exposure, and forward-secrecy should have made revealing Bob's state useless.[2]

Overview We follow in the footsteps of [BSJ+17] and study RKE as a general cryptographic primitive. However, we significantly improve on their results, in three independent directions:

Firstly, we extend the strictly unidirectional RKE concept of Bellare et al. towards bidirectional communication. In more detail, if we refer to the setting of [BSJ+17] as URKE (unidirectional RKE), we introduce SRKE (sesquidirectional[3] RKE) and BRKE (bidirectional RKE). In SRKE, while both Alice and Bob can send ciphertexts to the respective peer, only the ciphertexts sent from Alice to Bob establish session keys. Those sent by Bob have no direct functionality but may

[2] A protocol that achieves security in the described setting is developed in this chapter; the central idea behind our construction is that Bob's key pair (y, Y) does not stay fixed but is updated each time a ciphertext is processed.

[3] Recall that 'sesqui' is Latin for one-and-a-half.

help him healing from state exposure. Also in BRKE both parties send ciphertexts, but here the situation is symmetric in that all ciphertexts establish keys (plus allow for healing from state exposure). As fully bidirectional RKE is the ultimate goal, URKE and SRKE introduce the necessary building blocks—both regarding the security model and the instantiation. Consequently we introduce them one after another. A conceptual and syntactical overview over all three primitives is in Figure 3.1.

Secondly, we propose an improved security model for URKE, and introduce security models for SRKE and BRKE. Our bidirectional models assume the likely only practical communication setting for messaging protocols, namely the one in which the operations of both parties can happen concurrently (in contrast to, say, according to a ping-pong pattern). We develop our models following a purist approach: We start with giving the adversary the full set of options to undertake its attack (including state exposures of *both* parties), and then exclude, one by one, those configurations that unavoidably lead to a 'trivial win' (an example for the latter is if the adversary first compromises Bob's state and then correctly 'guesses' the next session key he recovers from an incoming ciphertext). This approach leads to strong and convincing security models (and it becomes quite challenging to actually meet them). We note that the (as we argued) insecure protocol from [BSJ+17] is considered secure in the model of [BSJ+17] because the latter was not designed with our strategy in mind, ultimately missing some attacks.

Thirdly, we give provably secure constructions of URKE, SRKE, and BRKE. While all prior RKE protocol proposals, including the one from [BSJ+17], are explicitly based on DHKE as a low-level tool, our constructions use generic primitives like KEMs, MACs, one-time signatures, and random oracles. The increased level of abstraction not only clarifies on the role that these components play in the constructions, it also increases the freedom when picking acceptable hardness assumptions.

Further details on our URKE construction. In brief, our

(unidirectional) URKE scheme combines a hash chain and KEM encapsulations to achieve both forward-secrecy and recoverability from state exposures. The crucial difference to the protocol from [BSJ+17] is that in our scheme the public key of Bob is changed after each use. Concretely, but omitting many details, the state information of Alice is $(i, k.p, Y)$ as in [BSJ+17] (but where Y is the *current* public key of Bob), for sending Alice freshly encapsulates a key k^* to Y, then computes $(k, k.p', k.u') \leftarrow \mathrm{H}(i, k.p, Y, k^*)$ using a random oracle H, and finally uses auxiliary key $k.u'$ to update the old public key Y to a new public key Y' that is to be used in her next sending operation. Bob does correspondingly, updating his secret key with each incoming ciphertext. Note that the attack against [BSJ+17] that we sketched above does not work against this protocol (the adversary would obtain a useless decryption key when revealing Bob's state).

Further details on our SRKE construction. Recall that, in SRKE, Bob can send update ciphertexts to Alice with the idea that this will help him to recover from state exposures. Our protocol algorithms can handle fully concurrent operation of the two participants (in particular, ciphertexts may 'cross' on the wire). This unfortunately adds, as the algorithms need to handle multiple 'epochs' (i.e., execution slots divided by send operations of Bob) at the same time, considerably to their complexity. Interestingly, the more involved communication setting is also reflected in stronger primitives that we require for our construction: Our SRKE construction builds on a special KEM type that supports so-called key updates (also the latter primitive is constructed in this chapter, from HIBE).

In a nutshell, in our SRKE construction, Bob heals from state exposures by generating a fresh (updatable) KEM key pair every now and then, and communicating the public key to Alice. Alice uses the key update functionality to 'fast-forward' these keys into a current state by making them aware of ciphertexts that were exchanged after the keys were sent (by Bob), but before they were received (by Alice). In her following sending operation, Alice encapsulates to a mix of old and new public keys.

Further details on our BRKE construction. We have two BRKE constructions. The first works via the amalgamation of two generic SRKE instances, deployed in reverse directions. To reach full security, the instances need to be carefully tied together, which we do via one-time signatures (akin to the CHK transform [MRY04, CHK04]). The second construction is less generic, namely by combining and interleaving the building blocks of our SRKE scheme in the right way. The advantage of the second scheme is that its ciphertexts are shorter (it saves precisely the one-time signatures).

Introducing SKRE as a natural building block for BRKE is consequently of particular value.

Related Constructions and Analyses The idea of mixing into the user state of messaging protocols additional key material that is continuously established with asymmetric techniques (in particular: DHKE) first appeared in the off-the-record (OTR) messaging protocol from [OTR16, BGB04] and thereafter in further protocols like the ZRTP telephony protocol [ZJC11] and the *Double Ratchet Algorithm* [PM16] (formerly known as Axolotl).

For the formal analysis of this latter protocol, Cohn-Gordon et al. [CCD+17] develop a "*model with adversarial queries and freshness conditions that capture the security properties intended by Signal*". Consequently, this model is not (primarily) aimed to serve as a reference notion for RKE. Also the line of multi-stage key exchange model frameworks by Fischlin and Günther [FG14, DFGS15, FG17], that served as a basis for the model used by Cohn-Gordon et al. [CCD+17], generally considers state updates and the continuous agreement of authenticated keys. Therefore, when compared to our work, these frameworks first and foremost can be seen as an alternative style of formally defining security since neither of them specifically considers security of RKE. In contrast, Cohn-Gordon et al. [CCG16] survey approaches in the literature for security of authenticated key agreement in the face of state exposures and propose a specific model.

3.2 Key-updatable Key Encapsulation Mechanisms

We introduce a type of KEM that we refer to as key-updatable. Like a regular KEM the new primitive establishes secure symmetric session keys, but in addition a dedicated key-update algorithm updates the components of the asymmetric key pair: Also taking an auxiliary input into account that we call the associated data, a secret key is updated to a new secret key, or a public key is updated to a new public key. A KEM key pair remains functional under such updates, meaning that symmetric session keys encapsulated for the public key can be recovered using the secret key if both (asymmetric) keys are updated compatibly, i.e., with matching associated data. Concerning security we require a kind of forward-secrecy: Briefly, session keys encapsulated to a (potentially updated) public key shall remain secure even if the adversary gets hold of any incompatibly updated version of the secret key. The syntactical concept of key-updatable KEM is shown in Figure 3.2.

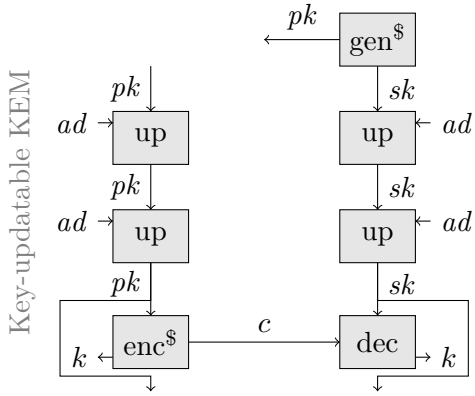

Figure 3.2: Conceptual depiction of kuKEM. '$' in the upper index of an algorithm name denotes that the algorithm runs probabilistically and *ad* is associated data.

This stronger variant of KEM is used as a core building block for

our SRKE and BRKE constructions in sections 3.6 and 3.9. Since our instantiation of this building block, introduced below, is relatively inefficient—implying inefficiency of our SRKE and BRKE constructions—we specifically analyze in Chapter 4 under which conditions key-updatable KEMs are indeed necessary and sufficient to realize secure ratcheted key exchange.

A *key-updatable key encapsulation mechanism* (kuKEM) for a finite session-key space $\mathcal{K}$ is a quadruple $\mathsf{K} = (\text{gen}_\mathsf{K}, \text{enc}, \text{dec}, \text{up})$ of algorithms together with a samplable secret-key space $\mathcal{SK}$, a public-key space $\mathcal{PK}$, a ciphertext space $\mathcal{C}$, and an associated-data space $\mathcal{AD}$. Algorithms $\text{gen}_\mathsf{K}, \text{enc}, \text{dec}$ are as for regular KEMs. The key-update algorithm up is deterministic and comes in two shapes: either it takes a secret key $sk \in \mathcal{SK}$ and associated data $ad \in \mathcal{AD}$ and outputs an updated secret key $sk' \in \mathcal{SK}$, or it takes a public key $pk \in \mathcal{PK}$ and associated data $ad \in \mathcal{AD}$ and outputs an updated public key $pk' \in \mathcal{PK}$. Shortcut notations for the key update algorithm(s) are thus

$$\mathcal{SK} \times \mathcal{AD} \to \text{up} \to \mathcal{SK} \qquad \mathcal{PK} \times \mathcal{AD} \to \text{up} \to \mathcal{PK} \ .$$

For correctness we require that for all $(sk_0, pk_0) \in [\text{gen}_\mathsf{K}]$ and $ad_1, \ldots, ad_n \in \mathcal{AD}$, if we let $sk_i = \text{up}(sk_{i-1}, ad_i)$ and $pk_i = \text{up}(pk_{i-1}, ad_i)$ for all i, then for all $(k, c) \in [\text{enc}(pk_n)]$ we have $\text{dec}(sk_n, c) = k$.

As a security property for kuKEMs we formalize a multi-receiver/multi-challenge version of one-way security that also reflects forward-secrecy in case of secret-key updates. The details of the notion are in game KUOW in Figure 3.3. For a key-updatable KEM K, we associate with any adversary $\mathcal{A}$ its one-way advantage $\text{Adv}^{\text{kuow}}_\mathsf{K}(\mathcal{A}) := \Pr[\text{KUOW}_\mathsf{K}(\mathcal{A}) \Rightarrow 1]$. Intuitively, a kuKEM is secure if all practical adversaries have a negligible advantage.

We extend our definition for regular KEMs by allowing the adversary to also update the public keys held by senders (encryptors) and the secret keys held by receivers, such that if a sender performs s-many updates using associated data from the list $ads = ad_1 \,\|\, \ldots \,\|\, ad_s$, a receiver performs r-many updates using associated data from the list $adr = ad'_1 \,\|\, \ldots \,\|\, ad'_r$, and the receiver is then exposed, then session

Game $\text{KUOW}_{\mathsf{K}}(\mathcal{A})$
00 $n \leftarrow 0$
01 Invoke $\mathcal{A}$
02 Stop with 0

Oracle Gen
03 $n \leftarrow n + 1$
04 $(sk_n, pk_n) \leftarrow_{\$} \text{gen}_{\mathsf{K}}$
05 $\text{CK}_n[\cdot] \leftarrow \bot$; $\text{XP}_n \leftarrow \emptyset$
06 $ads_n \leftarrow \epsilon$; $adr_n \leftarrow \epsilon$
07 $\text{SK}_n[\cdot] \leftarrow \bot$
08 $\text{SK}_n[adr_n] \leftarrow sk_n$
09 Return pk_n

Oracle $\text{Up}_S(i, ad)$
10 Require $1 \le i \le n$
11 $pk_i \leftarrow \text{up}(pk_i, ad)$
12 $ads_i \xleftarrow{\shortparallel} ad$
13 Return pk_i

Oracle Enc(i)
14 Require $1 \le i \le n$
15 $(k, c) \leftarrow_{\$} \text{enc}(pk_i)$
16 $\text{CK}_i[ads_i, c] \leftarrow k$
17 Return c

Oracle Solve(i, ad, c, k)
18 Require $1 \le i \le n$
19 Require $ad \notin \text{XP}_i$
20 Require $\text{CK}_i[ad, c] \neq \bot$
21 Reward $k = \text{CK}_i[ad, c]$
22 Return

Oracle $\text{Up}_R(i, ad)$
23 Require $1 \le i \le n$
24 $sk_i \leftarrow \text{up}(sk_i, ad)$
25 $adr_i \xleftarrow{\shortparallel} ad$
26 $\text{SK}_i[adr_i] \leftarrow sk_i$
27 Return

Oracle Check(i, ad, c, k)
28 Require $1 \le i \le n$
29 Require $\text{SK}_i[ad] \neq \bot$
30 $k' \leftarrow \text{dec}(\text{SK}_i[ad], c)$
31 Return $[k' = k]$

Oracle Expose(i)
32 Require $1 \le i \le n$
33 $\text{XP}_i \xleftarrow{\cup} \{A \in \mathcal{AD}^* : adr_i \preceq A\}$
34 Return sk_i

Figure 3.3: Security experiment KUOW, modeling the one-way security of a key-updatable KEM in a multi-receiver/multi-challenge setting. Oracles Up_S and Up_R update senders and receivers, respectively, and for each receiver i the lists ads_i and adr_i record the associated data used for updating the corresponding sender and receiver keys, respectively. See Figure 2.4 for an explanation of the other game variables. The instruction in line 33 adjoins to set XP_i the set of all adr_i-prefixed sequences of associated data.

keys encapsulated by the sender for the receiver remain hidden from the adversary if keys were updated inconsistently (with conflicting associated data, or too often on the receiver side), i.e., technically, if $adr \not\preceq ads$ (adr is a not a prefix of ads).

Constructing key-updatable KEMs. Observe that kuKEMs are related to hierarchical identity-based encryption (HIBE, [GS02]): Intuitively, updating a secret key using associated data ad in the kuKEM

world corresponds in the HIBE world with delegating the decryption/delegation key for the next-lower hierarchy level, using partial identity ad. Indeed, a kuKEM scheme is immediately constructed from a generic HIBE, with only cosmetic changes necessary when annotating the algorithms; a construction is provided in Figure 3.4. Also the security reduction from kuKEM to HIBE is immediate.

Proc gen_{K}
00 $(sk, pk) \leftarrow_{\$} \text{gen}_{\mathsf{HK}}$
01 $ad_0 \leftarrow \epsilon$
02 $sk \leftarrow_{\$} \text{del}_{\mathsf{HK}}(sk, ad_0)$
03 $id \leftarrow ad_0$; $pk \leftarrow (pk, id)$
04 Return (sk, pk)

Proc $\text{up}(pk, ad)$
05 $(pk', id) \leftarrow pk$
06 $pk \leftarrow (pk', id \,\|\, ad)$
07 Return pk

Proc $\text{enc}(pk)$
08 $(pk', id) \leftarrow pk$
09 $(k, c) \leftarrow_{\$} \text{enc}_{\mathsf{HK}}(pk', id)$
10 Return (k, c)

Proc $\text{dec}(sk, c)$
11 $k \leftarrow \text{dec}_{\mathsf{HK}}(sk, c)$
12 Return k

Proc $\text{up}(sk, ad)$
13 $sk \leftarrow \text{del}_{\mathsf{HK}}(sk, ad)$
14 Return sk

Figure 3.4: Construction of a key-updatable KEM from a generic HIBE. The delegation operation during key generation in line 02 is necessary as some HIBEs (for instance the Gentry–Silverberg scheme [GS02]) do not support encapsulating to the root node of the hierarchy; our construction thus does the first descent during key setup.

While HIBE schemes generically imply kuKEMs, it is unclear whether the same holds in the reverse direction, i.e., whether HIBEs can be constructed from kuKEMs. The crucial observation is that kuKEMs support only one strand of secret-key updates (recall our KUOW notion says nothing about what happens when a receiver duplicates its secret key and updates it twice, with different associated data), while HIBE schemes support at least two subidentities per node. Indeed, a (separating) example of a secure kuKEM that results in a weak HIBE when converted in the intuitive way is easily found.[4] Our conclusion is that while all kuKEM constructions we are aware of require HIBE

[4]For instance, conceptually, when updating a secret key to a new one, the kuKEM could secret-share the old key using a two-out-of-two threshold scheme, randomly pick one of the shares and include it in the new key, discarding the other share. While this would not hurt kuKEM security, a naively derived HIBE

and thus practically undesirable building blocks like pairings or lattices, kuKEMs seem to be a strictly weaker primitive than HIBE, so it is more likely to find constructions in the bare DLP setting. We leave it as an open problem to find such a construction.

3.3 Unidirectionally ratcheted key exchange (URKE)

We give a definition of unidirectional RKE and its security. While, in principle, our syntactical definition is in line with the one from [BSJ+17], our naming convention deviates significantly from the latter for the sake of a more clear distinction between (session) keys, (session) states, and ciphertexts[5] and we stress that, looking ahead, our security notion for URKE is—when only considering state exposures but disregarding the leakage of used randomness—stronger than the one of [BSJ+17]. A speciality of our formalization is that we let the sending and receiving algorithms of Alice and Bob accept and process an associated-data string [Rog02] that, for functionality, has to match on both sides.

A *unidirectionally ratcheted key exchange* (URKE) for a finite key space $\mathcal{K}$ and an associated-data space $\mathcal{AD}$ is a triple $\mathsf{R} = (\text{init}, \text{snd}, \text{rcv})$ of algorithms together with a sender state space $\mathcal{S}_A$, a receiver state space $\mathcal{S}_B$, and a ciphertext space $\mathcal{C}$. The randomized initialization algorithm init returns a sender state $st_A \in \mathcal{S}_A$ and a receiver state $st_B \in \mathcal{S}_B$. The randomized sending algorithm snd takes a state $st_A \in \mathcal{S}_A$ and an associated-data string $ad \in \mathcal{AD}$, and produces an updated state $st'_A \in \mathcal{S}_A$, a key $k \in \mathcal{K}$, and a ciphertext $c \in \mathcal{C}$. Finally, the deterministic receiving algorithm rcv takes a state $st_B \in \mathcal{S}_B$, an associated-data string $ad \in \mathcal{AD}$, and a ciphertext $c \in \mathcal{C}$, and either outputs an updated state $st'_B \in \mathcal{S}_B$ and a key $k \in \mathcal{K}$, or the spe-

would be trivial to break.

[5] The mapping between our names (on the left of the equality sign) and the ones of [BSJ+17] (on the right) is as follows: '(session) key' = 'output key', '(session) state' = 'session key plus sender/receiver key', 'ciphertext' = 'update information'.

cial symbol $\perp$ to indicate rejection. A shortcut notation for these syntactical definitions (conceptually shown in Figure 3.1) is

$$\begin{array}{rcccl} & & \text{init} & \to_{\$} & \mathcal{S}_A \times \mathcal{S}_B \\ \mathcal{S}_A \times \mathcal{AD} & \to & \text{snd} & \to_{\$} & \mathcal{S}_A \times \mathcal{K} \times \mathcal{C} \\ \mathcal{S}_B \times \mathcal{AD} \times \mathcal{C} & \to & \text{rcv} & \to & (\mathcal{S}_B \times \mathcal{K}) \cup \{(\perp, \perp)\} \end{array}$$

Correctness of URKE. Assume a sender and a receiver that were jointly initialized with init. Then, intuitively, the URKE scheme is correct if for all sequences (ad_i) of associated-data strings, if (k_i) and (c_i) are sequences of keys and ciphertexts successively produced by the sender on input the strings in (ad_i), and if (k'_i) is the sequence of keys output by the receiver on input the (same) strings in (ad_i) and the ciphertexts in (c_i), then the keys of the sender and the receiver match, i.e., it holds that $k_i = k'_i$ for all i.

We formalize this requirement via the FUNC game in Figure 3.5.[6] Concretely, we say scheme R is *correct* if $\Pr[\text{FUNC}_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1] = 0$ for all adversaries $\mathcal{A}$. In the game, the adversary lets the sender and the receiver process associated-data strings and ciphertexts of its choosing, and its goal is to let the two parties compute keys that do not match when they should. Variables s_A and r_B count the send and receive operations, associative array AC_A jointly records the associated-data strings considered by and the ciphertexts produced by the sender, flag is_B is an indicator that tracks whether the receiver is still 'in-sync' (in contrast to: was exposed to non-matching associated-data strings or ciphertexts; note how the transition between in-sync and out-of-sync is detected and recorded in lines 13,14), and associative array K_A records the keys established by the sender to allow for a comparison with the keys recovered (or not) by the receiver. The correctness requirement boils down to declaring the adversary successful

[6] Formalizing correctness of URKE via a game might at first seem overkill. However, for SRKE and BRKE, which allow for interleaved interaction in two directions, game-based definitions seem to be natural and notationally superior to any other approach. For consistency we use a game-based definition also for URKE.

(in line 17) if the sender and the receiver compute different keys while still being in-sync. Note finally that lines 12,16 ensure that once the rcv algorithm rejects, the adversary is notified of this and further queries to the RcvB oracle are not accepted.

Game $\text{FUNC}_{\mathsf{R}}(\mathcal{A})$	**Oracle** $\text{SndA}(ad)$	**Oracle** $\text{RcvB}(ad, c)$
00 $s_A \leftarrow 0$; $r_B \leftarrow 0$	07 $(st_A, k, c) \leftarrow_{\$} \text{snd}(st_A, ad)$	12 Require $st_B \neq \bot$
01 $\text{AC}_A[\cdot] \leftarrow \bot$	08 $\text{AC}_A[s_A] \leftarrow (ad, c)$	13 If $is_B \wedge \text{AC}_A[r_B] \neq (ad, c)$:
02 $is_B \leftarrow \mathtt{T}$	09 $\text{K}_A[s_A] \leftarrow k$	14 $\quad is_B \leftarrow \mathtt{F}$
03 $\text{K}_A[\cdot] \leftarrow \bot$	10 $s_A \leftarrow s_A + 1$	15 $(st_B, k) \leftarrow \text{rcv}(st_B, ad, c)$
04 $(st_A, st_B) \leftarrow_{\$} \text{init}$	11 Return c	16 If $st_B = \bot$: Return $\bot$
05 Invoke $\mathcal{A}$		17 Reward $is_B \wedge k \neq \text{K}_A[r_B]$
06 Stop with 0		18 $r_B \leftarrow r_B + 1$
		19 Return

Figure 3.5: Game FUNC for URKE scheme R.

Security of URKE. We formalize a key indistinguishability notion for URKE. In a nutshell, from the point of view of the adversary, keys established by the sender and recovered by the receiver shall look uniformly distributed in the key space. In our model, the adversary, in addition to scheduling the regular URKE operations via the SndA and RcvB oracles, has to its disposal the four oracles ExposeA, ExposeB, Reveal, and Challenge, used for exposing users by obtaining copies of their current state, for learning established keys, and for requesting real-or-random challenges on established keys, respectively. For an URKE scheme R, in Figure 3.6 we specify corresponding key indistinguishability games $\text{KIND}^b_{\mathsf{R}}$, where $b \in \{0, 1\}$ is the challenge bit, and we associate with any adversary $\mathcal{A}$ its key distinguishing advantage $\text{Adv}^{\text{kind}}_{\mathsf{R}}(\mathcal{A}) := |\Pr[\text{KIND}^1_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1] - \Pr[\text{KIND}^0_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1]|$. Intuitively, R offers key indistinguishability if all practical adversaries have a negligible key distinguishing advantage.

Most lines of code in the KIND^b games are tagged with a '$\cdot$' right after the line number; to the subset of lines marked in this way we refer to as the games' *core*. Conceptually, the cores contain all relevant game logic (participant initialization, specifications of how queries are

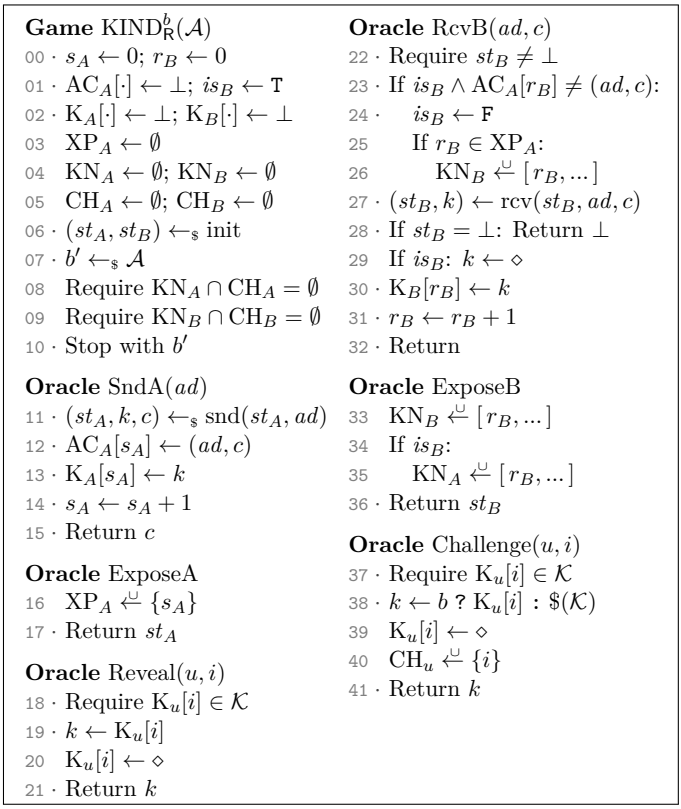

Figure 3.6: Games KIND^b, $b \in \{0, 1\}$, for URKE scheme R. We require $\diamond \notin \mathcal{K}$, and in Reveal and Challenge queries we require $u \in \{A, B\}$. If the notation in lines 26 or 38 is unclear, please consult Section 2.2.

answered, etc.); the code lines available only in the full game, i.e., the untagged ones, introduce certain restrictions on the adversary that are necessary to exclude trivial attacks (see below). The games' cores should be self-explanatory, in particular when comparing them to the FUNC game, with the understanding that lines 18,37 (in Figure 3.6) ensure that only keys can be revealed or challenged that actually have been established before, and that line 38 assigns to variable k, depending on bit b, either the real key or a freshly sampled element from the

key space.

Note that, in the pure core code, the adversary can use the four new oracles to bring itself into the position to distinguish real and random keys in a trivial way. In the following we discuss five different strategies to do so. We illustrate each strategy by specifying an example adversary in pseudocode and we explain what measures the full games take for disregarding the respective class of attack. (That is, the example adversaries would gain high advantage if the games consisted of just their cores, but in the full games their advantage is zero.)

The first two strategies leverage on the interplay of Reveal and Challenge queries; they do not involve exposing participants.

(a) The adversary requests a challenge on a key that it also reveals, it requests two challenges on the same key, or similar. Example: $c \leftarrow \text{SndA}(\epsilon)$; $k \leftarrow \text{Reveal}(A, 0)$; $k' \leftarrow \text{Challenge}(A, 0)$; $b' \leftarrow [k = k']$; output b'. The full games, in lines 20,39, overwrite keys that are revealed or challenged with the special symbol $\diamond \notin \mathcal{K}$. Because of lines 18,37, this prevents any second Reveal or Challenge query involving the same key.

(b) The adversary combines an attack from (a) with the correctness guarantee, i.e., that in-sync receivers recover the keys established by senders. For instance, the adversary reveals a sender key and requests a challenge on the corresponding receiver key. Example: $c \leftarrow \text{SndA}(\epsilon)$; $k \leftarrow \text{Reveal}(A, 0)$; $\text{RcvB}(\epsilon, c)$; $k' \leftarrow \text{Challenge}(B, 0)$; $b' \leftarrow [k = k']$; output b'. The full games, in line 29, overwrite in-sync receiver keys, as they are known (by correctness) to be the same on the sender side, with the special symbol $\diamond \notin \mathcal{K}$. By lines 18,37, this rules out the attack.

The remaining three strategies involve exposing participants and using their state to either trace their computations or impersonate them to their peer. In the full games, the set variables $\text{XP}_A, \text{KN}_A, \text{KN}_B, \text{CH}_A, \text{CH}_B$ (lines 03–05) help identifying when such attacks occur. Concretely, set XP_A tracks the points in time the sender is exposed (the

unit of time being the number of past sending operations; see line 16), sets $\mathrm{KN}_A, \mathrm{KN}_B$ track the indices of keys that are 'known' (in particular: traceable and recoverable) by the adversary using an exposed state (see below), and sets $\mathrm{CH}_A, \mathrm{CH}_B$ record the indices of keys for which a challenge was requested (see line 40). Lines 08,09 ensure that any adversary that requests to be challenged on a known and traceable key has advantage zero. Strategies (c) and (d) are state tracing attacks, while strategy (e) is based on impersonation.

(c) The adversary exposes the receiver and uses the obtained state to trace its computations: By iteratively applying the rcv algorithm to all later inputs of the receiver, and updating the exposed state correspondingly, the adversary implicitly obtains a copy of all later receiver keys. Example: $c \leftarrow \mathrm{SndA}(\epsilon)$; $st_B^* \leftarrow \mathrm{ExposeB}()$; $(st_B^*, k) \leftarrow \mathrm{rcv}(st_B^*, \epsilon, c)$; $\mathrm{RcvB}(\epsilon, c)$; $k' \leftarrow \mathrm{Challenge}(B, 0)$; $b' \leftarrow [k = k']$; output b'. When an exposure of the receiver happens, the full games, in line 33, mark all future receiver keys as known.

(d) The adversary combines the attack from (c) with the correctness guarantee, i.e., that in-sync receivers recover the keys established by senders: After exposing an in-sync receiver, by iteratively applying the rcv algorithm to all later outputs of the sender, the adversary implicitly obtains a copy of all later sender keys. Example: $c \leftarrow \mathrm{SndA}(\epsilon)$; $st_B^* \leftarrow \mathrm{ExposeB}()$; $(st_B^*, k) \leftarrow \mathrm{rcv}(st_B^*, \epsilon, c)$; $k' \leftarrow \mathrm{Challenge}(A, 0)$; $b' \leftarrow [k = k']$; output b'. When an exposure of an in-sync receiver happens, the full games, in lines 34,35, mark all future sender keys as known.

(e) Exposing the sender allows for impersonating it: The adversary obtains a copy of the sender's state and invokes the snd algorithm with it, obtaining a key and a ciphertext. The latter is provided to an in-sync receiver (rendering the latter out-of-sync), who recovers a key that is already known to the adversary. Example: $st_A^* \leftarrow \mathrm{ExposeA}()$; $(st_A^*, k, c) \leftarrow_{\$} \mathrm{snd}(st_A^*, \epsilon)$; $\mathrm{RcvB}(\epsilon, c)$; $k' \leftarrow \mathrm{Challenge}(B, 0)$; $b' \leftarrow [k = k']$; output b'. The full games, in

lines 25,26, detect the described type of impersonation and mark all future receiver keys as known.

We conclude with some notes on our URKE model. First, the model excludes the (anyway unavoidable) trivial attack conditions we identified, but nothing else. This establishes confidence in the model, as no attacks can be missed. Further, observe that it is not possible to recover from an attack based on state exposure (i.e., of the (c)–(e) types): If *one* key of a participant becomes weak as a consequence of a state exposure, then necessarily *all* later keys of that participant become weak as well. On the other hand, exposing the sender and *not* bringing the receiver out-of-sync does not affect security at all.[7] Finally, exposing an out-of-sync receiver does not harm later sender keys. In later sections we consider ratcheting primitives (SRKE, BRKE) that resume safe operation after state exposure attacks.

3.4 Constructing URKE

We construct an URKE scheme that is provably secure in the model presented in the previous section. The ingredients are a KEM (with deterministic public-key generation, see Chapter 2), a strongly unforgeable MAC, and a random oracle H. The algorithms of our scheme are specified in Figure 3.7.

We describe protocol states and algorithms in more detail. The state of Alice consists of (Bob's) KEM public key pk, a chaining key $k.c$, a MAC key $k.m$, and a transcript variable t that accumulates the associated-data strings and ciphertexts that Alice processed so far. The state of Bob is almost the same, but instead of the KEM public key he holds the corresponding secret key sk. Initially, sk and pk are freshly generated, random values are assigned to $k.c$ and $k.m$, and the transcript accumulator t is set to the empty string. A sending operation of Alice consists of invoking the KEM encapsulation routine

[7] This is precisely the distinguishing auto-recovery property of ratcheted key exchange.

with Bob's current public key, sampling and attaching a random symmetric 'collision' key, computing a MAC tag over the ciphertext and the associated data, updating the transcript accumulator, and jointly processing the session key established by the KEM, the chaining key, and the current transcript with the random oracle H. The output of H is split into the URKE session key $k.o$, an updated chaining key, an updated MAC key, and, indirectly, the updated public key (of Bob) to which Alice encapsulates in the next round. The receiving operation of Bob is analogue to these instructions. We add the collision key to the transmitted ciphertext only to avoid the use non-standard assumptions (such as plaintext-awareness) in our proof. While our scheme has some similarity with the one of [BSJ+17], a considerable difference is that the public and secret keys held by Alice and Bob, respectively, are constantly changed. This rules out the attack described in Section 3.1.

Proc init	**Proc** snd(st_A, ad)	**Proc** rcv(st_B, ad, C)
00 $(sk, pk) \leftarrow_{\$} \mathrm{gen}_{\mathsf{K}}$	06 $(pk, k.c, k.m, t) \leftarrow st_A$	16 $(sk, k.c, k.m, t) \leftarrow st_B$
01 $k.c \leftarrow_{\$} \mathcal{K}$; $k.m \leftarrow_{\$} \mathcal{K}$	07 $(k, c) \leftarrow_{\$} \mathrm{enc}(pk)$	17 $c \,\|\|\, ck \,\|\, \tau \leftarrow C$
02 $t \leftarrow \epsilon$	08 $ck \leftarrow_{\$} \mathcal{K}$	18 Require $\mathrm{vfy}_{\mathsf{M}}(k.m, ad \,\|\|\, c \,\|\, ck, \tau)$
03 $st_A \leftarrow (pk, k.c, k.m, t)$	09 $\tau \leftarrow \mathrm{tag}(k.m, ad \,\|\|\, c \,\|\, ck)$	19 $k \leftarrow \mathrm{dec}(sk, c)$
04 $st_B \leftarrow (sk, k.c, k.m, t)$	10 $C \leftarrow c \,\|\, ck \,\|\, \tau$	20 Require $k \neq \perp$
05 Return (st_A, st_B)	11 $t \xleftarrow{\shortparallel} ad \,\|\|\, C$	21 $t \xleftarrow{\shortparallel} ad \,\|\|\, C$
	12 $k.o \,\|\|\, k.c \,\|\|\, k.m \,\|\| \, sk \leftarrow \mathrm{H}(k.c, k, t)$	22 $k.o \,\|\|\, k.c \,\|\|\, k.m \,\|\| \, sk \leftarrow \mathrm{H}(k.c, k, t)$
	13 $pk \leftarrow \mathrm{gen}_{\mathsf{K}}(sk)$	23 $st_B \leftarrow (sk, k.c, k.m, t)$
	14 $st_A \leftarrow (pk, k.c, k.m, t)$	24 Return $(st_B, k.o)$
	15 Return $(st_A, k.o, C)$	

Figure 3.7: Construction of an URKE scheme from a key-encapsulation mechanism $\mathsf{K} = (\mathrm{gen}_{\mathsf{K}}, \mathrm{enc}, \mathrm{dec})$, a message authentication code $\mathsf{M} = (\mathrm{tag}, \mathrm{vfy}_{\mathsf{M}})$, and a random oracle H. For simplicity we denote the key space of the MAC and the space of chaining keys and collision keys with the same symbol $\mathcal{K}$.

Note that our scheme is specified such that participants accumulate in their state the full past communication history. While this eases the security analysis (random oracle evaluations of Alice and Bob are guaranteed to be on different inputs once the in-sync bit is cleared), it also seems to impose a severe implementation obstacle. However,

as current hash functions like SHA2 and SHA3 process inputs in an online fashion (i.e., left-to-right with a small state overhead), they can process append-only inputs like transcripts such that computations are efficiently shared with prior invocations. In particular, with such a hash function our URKE scheme can be implemented with constant-size state. (This requires, though, rearranging the input of H such that t comes first).[8]

Theorem 1 *The URKE protocol* R *from Figure 3.7 offers key indistinguishability. More precisely, if function* H *is modeled as a random oracle, for every adversary* $\mathcal{A}$ *against URKE scheme* R *in games* $\mathrm{KIND}^b_{\mathsf{R}}$ *from Figure 3.6 there exists an adversary* $\mathcal{B}_{\mathsf{K}}$ *against KEM* K *in game* OW *from Figure 2.4 and an adversary* $\mathcal{B}_{\mathsf{M}}$ *against MAC* M *in game* SUF *from Figure 2.2 such that* $\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{R}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{ow}}_{\mathsf{K}}(\mathcal{B}_{\mathsf{K}}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}}) + \frac{q_{\mathrm{H}}+1}{|\mathcal{K}|}$*, where* $\mathcal{K}$ *is the (collision-)key space, the running time of* $\mathcal{B}_{\mathsf{K}}$ *is about that of* $\mathcal{A}$ *plus* q_{H} *key checking and solve operations, the running time of* $\mathcal{B}_{\mathsf{M}}$ *is about that of* $\mathcal{A}$*, and* q_{H} *is the number of* $\mathcal{A}$*'s random oracle queries.*

For comprehensibility and didactic reasons we present the security proofs for our constructions of URKE, SRKE, and BRKE together at the end of this chapter. The proof of Theorem 1 is in Section 3.10. Briefly, it first shows that none of Alice's established session keys can be derived by the adversary without breaking the security of the KEM as long as no previous secret key of Alice's public keys was exposed. Then we show that Bob will only establish session keys out of sync if Alice was impersonated towards him, his state was exposed before, the adversary predicted a collision key, or a MAC forgery was conducted by the adversary. The latter will be reduced to the security of the MAC. Consequently the adversary either breaks one of the employed primitives' security or has information-theoretically small advantage in winning the KIND game.

[8] A different approach to achieve a constant-size state is to replace lines 11 and 21 by the (non-accumulating) assignments $t \leftarrow (ad, C)$. We believe our scheme would also be secure in this case as, intuitively, chaining key $k.c$ reflects the full past communication.

3.5 Sesquidirectionally ratcheted key exchange (SRKE)

We introduce *sesquidirectionally ratcheted key exchange* (SRKE)[3] as a generalization of URKE. The basic functionality of the two primitives is the same: Sessions involve two parties, A and B, where A can establish keys and safely share them with B by providing the latter with ciphertexts. In contrast to the URKE case, in SRKE also party B can generate and send ciphertexts (to A); however, B's invocations of the sending routine do not establish keys. Rather, the idea behind B communicating ciphertexts to A is that this may increase the security of the keys established by A. Indeed, as we will see, in SRKE it is possible for B to recover from attacks involving state exposure. We proceed with formalizing syntax and correctness of SRKE.

Formally, a SRKE scheme for a finite key space $\mathcal{K}$ and an associated-data space $\mathcal{AD}$ is a tuple $\mathsf{R} = (\text{init}, \text{snd}_A, \text{rcv}_B, \text{snd}_B, \text{rcv}_A)$ of algorithms together with a state space $\mathcal{S}_A$, a state space $\mathcal{S}_B$, and a ciphertext space $\mathcal{C}$. The randomized initialization algorithm init returns a state $st_A \in \mathcal{S}_A$ and a state $st_B \in \mathcal{S}_B$. The randomized sending algorithm snd_A takes a state $st_A \in \mathcal{S}_A$ and an associated-data string $ad \in \mathcal{AD}$, and produces an updated state $st'_A \in \mathcal{S}_A$, a key $k \in \mathcal{K}$, and a ciphertext $c \in \mathcal{C}$. The deterministic receiving algorithm rcv_B takes a state $st_B \in \mathcal{S}_B$, an associated-data string $ad \in \mathcal{AD}$, and a ciphertext $c \in \mathcal{C}$, and outputs either an updated state $st'_B \in \mathcal{S}_B$ and a key $k \in \mathcal{K}$, or the special symbol $\perp$ to indicate rejection. The randomized sending algorithm snd_B takes a state $st_B \in \mathcal{S}_B$ and an associated-data string $ad \in \mathcal{AD}$, and produces an updated state $st'_B \in \mathcal{S}_B$ and a ciphertext $c \in \mathcal{C}$. Finally, the deterministic receiving algorithm rcv_A takes a state $st_A \in \mathcal{S}_A$, an associated-data string $ad \in \mathcal{AD}$, and a ciphertext $c \in \mathcal{C}$, and outputs either an updated state $st'_A \in \mathcal{S}_A$ or the special symbol $\perp$ to indicate rejection. A shortcut notation for these syntactical definitions (conceptually shown in Figure 3.1) is

$$
\begin{array}{rcccl}
 & & \text{init} & \to_{\$} & \mathcal{S}_A \times \mathcal{S}_B \\
\mathcal{S}_A \times \mathcal{AD} & \to & \text{snd}_A & \to_{\$} & \mathcal{S}_A \times \mathcal{K} \times \mathcal{C} \\
\mathcal{S}_B \times \mathcal{AD} \times \mathcal{C} & \to & \text{rcv}_B & \to & (\mathcal{S}_B \times \mathcal{K}) \cup \{(\bot, \bot)\} \\
\mathcal{S}_B \times \mathcal{AD} & \to & \text{snd}_B & \to_{\$} & \mathcal{S}_B \times \mathcal{C} \\
\mathcal{S}_A \times \mathcal{AD} \times \mathcal{C} & \to & \text{rcv}_A & \to & \mathcal{S}_A \cup \{\bot\} \quad .
\end{array}
$$

Correctness of SRKE. Our definition of SRKE functionality is via game FUNC in Figure 3.8. We say scheme R is *correct* if $\Pr[\text{FUNC}_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1] = 0$ for all adversaries $\mathcal{A}$. In the figure, the lines of code tagged with a '$\cdot$' right after the line number also appear in the URKE FUNC game (Figure 3.5). In comparison with that game, there are two more oracles, SndB and RcvA, and four new game variables, $s_B, r_A, \text{AC}_B, is_A$, that control and monitor the communication in the B-to-A direction akin to how SndA, RcvB, $s_A, r_B, \text{AC}_A, is_B$ do (like in the URKE case) for the A-to-B direction. In particular, the is_A flag is the in-sync indicator of party A that tracks whether the latter was exposed to non-matching associated-data strings or ciphertexts (the transition between in-sync and out-of-sync is detected and recorded in lines 35,36). Given that the specifications of oracles SndA and RcvB of figures 3.5 and 3.8 coincide (with one exception: lines 13,21 are guarded by in-sync checks (in lines 12,20) so that parties go out-of-sync not only when processing unauthentic associated data or ciphertexts, but also when they process ciphertexts that were generated by an out-of-sync peer[9]), and that also the specifications of oracles SndB and RcvA of figures 3.8 are quite similar to them (besides the reversion of the direction of communication, the difference is that all session-key related components were stripped off), the logics of the FUNC game in Figure 3.8 should be clear. Overall, like in the URKE case, the correctness requirement boils down to declaring the adversary successful, in line 31, if A and B compute different keys while still being in-sync.

Epochs. The intuition behind having the B-to-A direction of communication in SRKE is that it allows B to refresh his state every now

[9] This approach is borrowed from [MP17, EMP18].

Game $\text{FUNC}_{\mathsf{R}}(\mathcal{A})$
00 · $s_A \leftarrow 0$; $r_B \leftarrow 0$
01 $s_B \leftarrow 0$; $r_A \leftarrow 0$
02 $e_A \leftarrow 0$; $\text{EP}_A[\cdot] \leftarrow \bot$
03 $E_B^{\lhd} \leftarrow 0$; $E_B^{\rhd} \leftarrow 0$
04 · $\text{AC}_A[\cdot] \leftarrow \bot$; $is_B \leftarrow \mathtt{T}$
05 $\text{AC}_B[\cdot] \leftarrow \bot$; $is_A \leftarrow \mathtt{T}$
06 · $\text{K}_A[\cdot] \leftarrow \bot$
07 · $(st_A, st_B) \leftarrow_{\$} \text{init}$
08 · Invoke $\mathcal{A}$
09 · Stop with 0

Oracle $\text{SndA}(ad)$
10 Require $st_A \neq \bot$
11 · $(st_A, k, c) \leftarrow_{\$} \text{snd}_A(st_A, ad)$
12 If is_A:
13 · $\text{AC}_A[s_A] \leftarrow (ad, c)$
14 $\text{EP}_A[s_A] \leftarrow e_A$
15 · $\text{K}_A[s_A] \leftarrow k$
16 · $s_A \leftarrow s_A + 1$
17 · Return c

Oracle $\text{SndB}(ad)$
18 Require $st_B \neq \bot$
19 $(st_B, c) \leftarrow_{\$} \text{snd}_B(st_B, ad)$
20 If is_B:
21 $\text{AC}_B[s_B] \leftarrow (ad, c)$
22 $E_B^{\rhd} \leftarrow E_B^{\rhd} + 1$
23 $s_B \leftarrow s_B + 1$
24 Return c

Oracle $\text{RcvB}(ad, c)$
25 · Require $st_B \neq \bot$
26 · If $is_B \wedge \text{AC}_A[r_B] \neq (ad, c)$:
27 · $is_B \leftarrow \mathtt{F}$
28 If is_B: $E_B^{\lhd} \leftarrow \text{EP}_A[r_B]$
29 · $(st_B, k) \leftarrow \text{rcv}_B(st_B, ad, c)$
30 · If $st_B = \bot$: Return $\bot$
31 · Reward $is_B \wedge k \neq \text{K}_A[r_B]$
32 · $r_B \leftarrow r_B + 1$
33 · Return

Oracle $\text{RcvA}(ad, c)$
34 Require $st_A \neq \bot$
35 If $is_A \wedge \text{AC}_B[r_A] \neq (ad, c)$:
36 $is_A \leftarrow \mathtt{F}$
37 If is_A: $e_A \leftarrow e_A + 1$
38 $st_A \leftarrow \text{rcv}_A(st_A, ad, c)$
39 If $st_A = \bot$: Return $\bot$
40 $r_A \leftarrow r_A + 1$
41 Return

Figure 3.8: Game FUNC for SRKE scheme R. The lines of code tagged with a '·' also appear in the URKE FUNC game. Note that the variables $e_A, \text{EP}_A, E_B^{\lhd}, E_B^{\rhd}$ do not influence the game outcome.

and then, and to inform A about this. The goal is to let B recover from state exposure.

Imagine, for example, a SRKE session where B has the following view on the communication: first he sends four refresh ciphertexts (to A) in a row; then he receives a key-establishing ciphertext (from A). As we assume a fully concurrent setting and do not impose timing constraints on the network delivery, the incoming ciphertext

can have been crafted by A after her having received (from B) between zero and four ciphertexts. That is, even though B refreshed his state a couple of times, to achieve correctness he has to remain prepared for recovering keys from ciphertexts that were generated by A before she recognized any of the refreshs. However, after processing A's ciphertext, if A created it after receiving some of B's ciphertexts (say, the first three), then the situation changes in that B is no longer required to process ciphertexts that refer to refreshs older than the one to which A's current answer is responding to (in the example: the first two).

These ideas turn out to be pivotal in the definition of SRKE security. We formalize them by introducing the notion of an *epoch*. Epochs start when the snd_B algorithm is invoked (each invocation starts one epoch), they are sequentially numbered, and the first epoch (with number zero) is implicitly started by the init algorithm. Each rcv_A invocation makes A aware of one new epoch, and subsequent snd_A invocations can be seen as occurring in its context. Finally, on B's side multiple epochs may be active at the same time, reflecting that B has to be ready to process ciphertexts that were generated by A in the context of one of potentially many possible epochs. Intuitively, epochs end (on B's side) if a ciphertext is received (from A) that was sent in the context of a later epoch.

We represent the span of epochs supported by B with the interval variable E_B (see Section 2.2): its boundaries $E_B^{\lhd}$ and $E_B^{\rhd}$ reflect at any time the earliest and the latest such epoch. Further, we use variable e_A to track the latest epoch started by B that party A is aware of, and associative array EP_A to register for each of A's sending operations the context, i.e., the epoch number that A is (implicitly) referring to. In more detail, the invocation of init is accompanied by setting $E_B^{\lhd}, E_B^{\rhd}, e_A$ to zero (in lines 02,03), each sending operation of B introduces one more supported epoch (line 22), each receiving operation of A increases the latter's awareness of epochs supported by B (line 37), the context of each sending operation of A is recorded in EP_A (line 14), and each receiving operation of B potentially reduces the number of supported epochs by dropping obsolete ones (line 28).

Observe that tracking epochs is not meaningful after participants get out-of-sync; we thus guard lines 37,28 with corresponding tests.

Security of SRKE. Our SRKE security model lifts the one for URKE to the bidirectional (more precisely: sesquidirectional) setting. The goal of the adversary is again to distinguish established keys from random. For a SRKE scheme R, the corresponding key indistinguishability games $\mathrm{KIND}^b_{\mathsf{R}}$, for challenge bit $b \in \{0,1\}$, are specified in Figure 3.9. With any adversary $\mathcal{A}$ we associate its key distinguishing advantage $\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{R}}(\mathcal{A}) := |\Pr[\mathrm{KIND}^1_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathrm{KIND}^0_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1]|$. Intuitively, R offers key indistinguishability if all practical adversaries have a negligible key distinguishing advantage.

The new KIND games are the natural amalgamation of the (URKE) KIND games of Figure 3.6 with the (SRKE) FUNC game of Figure 3.8 (with the exceptions discussed below). Concerning the trivial attack conditions on URKE that we identified in Section 3.3, we note that conditions (a) and (b) remain valid for SRKE without modification, conditions (c) and (d) (that consider attacks on participants by tracing their computations) need a slight adaptation to reflect that updating epochs repairs the damage of state exposures, and condition (e) (that considers impersonation of exposed A to B), besides needing a slight adaptation, requires to be complemented by a new condition that considers that exposing B allows for impersonating him to A.

When comparing the KIND games from figures 3.6 and 3.9, note that a crucial difference is that the $\mathrm{K}_A, \mathrm{K}_B$ arrays in the URKE model are indexed with simple counters, while in the SRKE model they are indexed with pairs where the one element is the same counter as in the URKE case and the other element indicates the epoch for which the corresponding key was established[10]. The new indexing mechanism allows, when B is exposed, for marking as known only those future keys of A and B that belong to the epochs managed by B

[10]The adversary always knows the epoch numbers associated with keys, so it can pose meaningful Reveal and Challenge queries just as before.

Game $\mathrm{KIND}^b_{\mathsf{R}}(\mathcal{A})$
00 · $s_A \leftarrow 0$; $r_B \leftarrow 0$
01 · $s_B \leftarrow 0$; $r_A \leftarrow 0$
02 · $e_A \leftarrow 0$; $\mathrm{EP}_A[\cdot] \leftarrow \bot$
03 · $E_B^{\lessdot} \leftarrow 0$; $E_B^{\gtrdot} \leftarrow 0$
04 · $\mathrm{AC}_A[\cdot] \leftarrow \bot$; $is_B \leftarrow \mathtt{T}$
05 · $\mathrm{AC}_B[\cdot] \leftarrow \bot$; $is_A \leftarrow \mathtt{T}$
06 $\mathrm{K}_A[\cdot] \leftarrow \bot$; $\mathrm{K}_B[\cdot] \leftarrow \bot$
07 $\mathrm{XP}_A \leftarrow \emptyset$; $\mathrm{XP}_B \leftarrow \emptyset$
08 $\mathrm{KN}_A \leftarrow \emptyset$; $\mathrm{KN}_B \leftarrow \emptyset$
09 $\mathrm{CH}_A \leftarrow \emptyset$; $\mathrm{CH}_B \leftarrow \emptyset$
10 · $(st_A, st_B) \leftarrow_{\$}$ init
11 · $b' \leftarrow_{\$} \mathcal{A}$
12 Require $\mathrm{KN}_A \cap \mathrm{CH}_A = \emptyset$
13 Require $\mathrm{KN}_B \cap \mathrm{CH}_B = \emptyset$
14 · Stop with b'

Oracle $\mathrm{SndA}(ad)$
15 · Require $st_A \neq \bot$
16 · $(st_A, k, c) \leftarrow_{\$} \mathrm{snd}_A(st_A, ad)$
17 · If is_A:
18 · $\quad \mathrm{AC}_A[s_A] \leftarrow (ad, c)$
19 · $\quad \mathrm{EP}_A[s_A] \leftarrow e_A$
20 $\mathrm{K}_A[e_A, s_A] \leftarrow k$
21 · $s_A \leftarrow s_A + 1$
22 · Return c

Oracle $\mathrm{RcvA}(ad, c)$
23 · Require $st_A \neq \bot$
24 · If $is_A \wedge \mathrm{AC}_B[r_A] \neq (ad, c)$:
25 · $\quad is_A \leftarrow \mathtt{F}$
26 $\quad$ If $r_A \in \mathrm{XP}_B$:
27 $\quad\quad \mathrm{KN}_A \xleftarrow{\cup} \mathbb{N} \times [\, s_A, \ldots]$
28 · If is_A: $e_A \leftarrow e_A + 1$
29 · $st_A \leftarrow \mathrm{rcv}_A(st_A, ad, c)$
30 · If $st_A = \bot$: Return $\bot$
31 · $r_A \leftarrow r_A + 1$
32 · Return

Oracle $\mathrm{RcvB}(ad, c)$
33 · Require $st_B \neq \bot$
34 · If $is_B \wedge \mathrm{AC}_A[r_B] \neq (ad, c)$:
35 · $\quad is_B \leftarrow \mathtt{F}$
36 $\quad$ If $r_B \in \mathrm{XP}_A$:
37 $\quad\quad \mathrm{KN}_B \xleftarrow{\cup} \mathbb{N} \times [\, r_B, \ldots]$
38 · If is_B: $E_B^{\lessdot} \leftarrow \mathrm{EP}_A[r_B]$
39 · $(st_B, k) \leftarrow \mathrm{rcv}_B(st_B, ad, c)$
40 · If $st_B = \bot$: Return $\bot$
41 If is_B: $k \leftarrow \diamond$
42 $\mathrm{K}_B[E_B^{\lessdot}, r_B] \leftarrow k$
43 · $r_B \leftarrow r_B + 1$
44 · Return

Oracle $\mathrm{SndB}(ad)$
45 · Require $st_B \neq \bot$
46 · $(st_B, c) \leftarrow_{\$} \mathrm{snd}_B(st_B, ad)$
47 · If is_B:
48 · $\quad \mathrm{AC}_B[s_B] \leftarrow (ad, c)$
49 · $\quad E_B^{\gtrdot} \leftarrow E_B^{\gtrdot} + 1$
50 · $s_B \leftarrow s_B + 1$
51 · Return c

Oracle ExposeA
52 If is_A: $\mathrm{XP}_A \xleftarrow{\cup} \{s_A\}$
53 Return st_A

Oracle ExposeB
54 $\mathrm{KN}_B \xleftarrow{\cup} [E_B^{\lessdot} \,..\, E_B^{\gtrdot}] \times [\, r_B, \ldots]$
55 If is_B:
56 $\quad \mathrm{XP}_B \xleftarrow{\cup} \{s_B\}$
57 $\quad \mathrm{KN}_A \xleftarrow{\cup} [E_B^{\lessdot} \,..\, E_B^{\gtrdot}] \times [\, r_B, \ldots]$
58 Return st_B

Oracle $\mathrm{Reveal}(u, i)$
as in URKE (Fig. 3.6)

Oracle $\mathrm{Challenge}(u, i)$
as in URKE (Fig. 3.6)

Figure 3.9: Games KIND^b, $b \in \{0, 1\}$, for SRKE scheme R. Lines of code tagged with a '·' similarly appear in the SRKE FUNC game in Figure 3.8.

at the time of exposure (lines 54,57). This already implements the necessary adaptation of conditions (c) and (d) to the SRKE setting. The announced adaptation of condition (e) is executing line 52 only if $is_A = \mathtt{T}$; the change is due as the motivation given in Section 3.3 is valid only if A is in-sync (which is always the case in URKE, but not in SRKE). Finally, complementing condition (e), we identify the following new trivial attack condition:

(f) Exposing party B allows for impersonating it: Assume parties A and B are in-sync. The adversary obtains a copy of B's state and invokes the snd_B algorithm with it, obtaining a ciphertext which it provides to party A (rendering the latter out-of-sync). If then A generates a new key using the snd_A algorithm, the adversary can feed the resulting ciphertext into the rcv_B algorithm, recovering the key. Example: $st_B^* \leftarrow \mathrm{ExposeB}()$; $(st_B^*, c) \leftarrow_{\$} \mathrm{snd}_B(st_B^*, \epsilon)$; $\mathrm{RcvA}(\epsilon, c)$; $c' \leftarrow \mathrm{SndA}(\epsilon)$; $(st_B^*, k) \leftarrow \mathrm{rcv}_B(st_B^*, \epsilon, c')$; $k' \leftarrow \mathrm{Challenge}(A, 0)$; $b' \leftarrow [k = k']$; output b'. Lines 26,27 (in conjunction with lines 07,56) detect the described type of impersonation and mark all future keys of A as known.

This completes the description of our SRKE security model. As in URKE, it excludes the minimal set of attacks, indicating that it gives strong security guarantees.

3.6 Constructing SRKE

We present a SRKE construction that generalizes our URKE scheme to the sesquidirectional setting. The core intuition is as follows: Like in the URKE scheme, A-to-B ciphertexts correspond with KEM ciphertexts where the corresponding public and secret keys are held by A and B, respectively, and the two keys are evolved to new keys after each use. In addition to this, with the goal of letting B heal from state exposures, our SRKE construction gives him the option to sanitize his state by generating a fresh KEM key pair and communicating the corresponding public key to A (using the B-to-A link specific to

SRKE). The algorithms of our protocol are specified in Figure 3.10. Although the sketched approach might sound simple and natural, the algorithms, quite surprisingly, are involved and require strong cryptographic building blocks (a key-updatable KEM and a one-time signature scheme, see Section 3.2 and Chapter 2). Their complexity is a result of SRKE protocols having to simultaneously offer solutions to multiple inherent challenges. We discuss these in the following.

Epoch management. Recall that we assume a concurrent setting for SRKE and that, thus, if B refreshes his state via the snd_B algorithm, then he still has to keep copies of the secret keys maintained for prior epochs (so that the rcv_B algorithm can properly process incoming ciphertexts created for them). Our protocol algorithms implement this by including in B's state the array $SK[\cdot]$ in which snd_B stores all keys it generates (line 26; obsolete keys of expired epochs are deleted by rcv_B in line 46). Beyond that, both A and B maintain an interval variable E in their state: its boundaries $E^{\lhd}$ and $E^{\rhd}$ are used by B to reflect the earliest and latest supported epoch, and by A to keep track of epoch updates that occur in direct succession (i.e., that are still waiting for their 'activation' by snd_A). Note finally that the snd_A algorithm communicates to rcv_B in every outgoing ciphertext the epoch in which A is operating (line 11).

Secure state update. Assume A executes once the snd_A algorithm, then twice the rcv_A algorithm, and then again once the snd_A algorithm. That is, following the above sketch of our protocol, as part of her first snd_A invocation she will encapsulate to a public key that she subsequently updates (akin to how she would do in our URKE solution, see lines 07,13 of Figure 3.7), then she will receive two fresh public keys from B, and finally she will again encapsulate to a public key that she subsequently updates. The question is: Which public key shall she use in the last step? The one resulting from the update during her first snd_A invocation, the one obtained in her first rcv_A invocation, or the one obtained in her second rcv_A invocation? We found that only one configuration is safe against key distinguishing attacks: Our SRKE protocol is such that she encapsulates to all three,

combining the established session keys into one via concatenation.[11] (We discuss why it is unsafe to encapsulate to only a subset of the keys in Section 3.7.3.) The algorithms implement this by including in A's state the array $PK[\cdot]$ in which rcv_A stores incoming public keys (line 60) and which snd_A consults when establishing outgoing ciphertexts (lines 12–14; the counterpart on B's side is in lines 39–43). Once the switch to the new epoch is completed, the obsolete public keys are removed from A's state (line 19). If A executes snd_A many times in succession, then all but the first invocation will, akin to the URKE case, just encapsulate to the (one) evolved public key from the preceding invocation.

We discuss a second issue related to state updates. Assume B executes three times the snd_B algorithm and then once the rcv_B algorithm, the latter on input a well-formed but non-authentic ciphertext (e.g., the adversary could have created the ciphertext, after exposing A's state, using the snd_A algorithm). In the terms of our security model the latter action brings B out-of-sync, which means that if he is subsequently exposed then this should not affect the security of further session keys established by A. On the other hand, according to the description provided so far, exposing B's state means obtaining a copy of array $SK[\cdot]$, i.e., of the decapsulation keys of all epochs still supported by B. We found that this easily leads to key distinguishing attacks,[12] so in order to protect the elements of $SK[\cdot]$ they are evolved by the rcv_B algorithm whenever an incoming ciphertext is processed. We implement the latter via the dedicated update procedure up provided by the key-updatable KEM. The corresponding lines are 47–48 (note that t^* is the current transcript fragment, see line 33). Of course A has to synchronize on B's key updates, which she does in lines 58–59, where array $L[\cdot]$ is the state variable that keeps track of the corresponding past A-to-B transcript fragments. (Outgoing ciphertexts are stored in $L[\cdot]$ in line 20, and obsolete ones are removed

[11] The concatenation of keys of an OW secure KEM can be seen as the implementation of a secure combiner in the spirit of [GHP18].

[12] We discuss this further in Section 3.7.2.

from it in line 57.) Note that A, for staying synchronized with B, also needs to keep track of the ciphertexts that he received (from her) so far; for this reason, B indicates in every outgoing ciphertext the number r of incoming ciphertexts he has been exposed to (lines 55,27).

Transcript management. Recall that one element of the participants' state in our URKE scheme (in Figure 3.7) is the variable t that accumulates transcript information (associated data, collision keys, and ciphertexts) of prior communication so that it can be input to key derivation. This is a common technique to ensure that the keys established on the two sides start diverging in the moment an active attack occurs. Also our SRKE construction follows this approach, but accumulating transcripts is more involved if communication is concurrent: If both A and B would add outgoing ciphertexts to their transcript accumulator directly after creating them, then concurrent sending would immediately desynchronize the two parties. This issue is resolved in our construction as follows: In the B-to-A direction, while A appends incoming ciphertexts (from B) to her transcript variable in the moment she receives them (line 53), when creating the ciphertexts, B will just record them in his state variable $L[\cdot]$ (line 29), and postpone adding them to his transcript variable to the point when he is able to deduce (from A's ciphertexts) the position of when she did (line 37; obsolete entries are removed in line 38). The A-to-B direction is simpler[13] and handled as in our URKE protocol: A updates her transcript when sending a ciphertext (line 16), and B updates his transcript when receiving it (lines 33,44). Note we tag transcript fragments with labels $\triangleright$ or $\triangleleft$ to indicate whether they emerged in the A-to-B or B-to-A direction of communication (e.g., in lines 16,29).

Authentication. To reach security against active adversaries we protect the SRKE ciphertexts against manipulation. Recall that in our URKE scheme a MAC (plus a random collision key) was sufficient for this. In SRKE, this is still sufficient for the A-to-B direction (lines 15,34), but for the B-to-A direction, to defend against attacks where

[13] Intuitively the disbalance comes from the fact that keys are only established by A-to-B ciphertexts and that transcripts are only used for key derivation.

the adversary first exposes A's state and then uses the obtained MAC key to impersonate B to her,[14] we need to employ a one-time signature scheme: Each ciphertext created by B includes a freshly generated verification key that is used to authenticate the next B-to-A ciphertext (lines 25,27,28,54,55; note how this rules out the described attack).

The only lines we did not comment on are 17,18,45,24—those that also form the core of our URKE protocol (which are discussed extensively in Section 3.4). A more detailed discussion of some design choices and more insights into insecure alternative constructions are in Section 3.7.

Practicality of our construction We remark that the number of updates per kuKEM key pair is bounded by the number of ciphertexts sent by A during one round-trip time (RTT) on the network between A and B (intuitively by the number of ciphertexts sent by A that *cross* the wire with one epoch update ciphertext from B). Ciphertexts that B did not know of when proposing an epoch (1/2 RTT) and ciphertexts A sent until she received the epoch proposal (1/2 RTT) are regarded for an update of a key pair. As a result, the hierarchy depth of an HIBE can be bounded by this number of ciphertexts when used for building a kuKEM for SRKE.

We further note that only the first and last send operation of A in an epoch involve kuKEM ciphertexts and all other encapsulations during A's sending in an epoch can base on only a one-way secure standard KEM. We emphasize this in our proof in Game 5.4 (see Section 3.11) but neglect this detail for simplicity.

[14] Note this is not an issue in the A-to-B direction: Exposing B and impersonating A to him leads to marking all future keys of B as known anyway, without any option to recover. We expand on this in Section 3.7.1.

Proc init
00 $(sgk, vfk) \leftarrow_{\$} \mathrm{gen}_{\mathsf{S}}$
01 · $(sk, pk) \leftarrow_{\$} \mathrm{gen}_{\mathsf{K}}$
02 · $k.c \leftarrow_{\$} \mathcal{K}$; $k.m \leftarrow_{\$} \mathcal{K}$; $t \leftarrow \epsilon$
03 $E^{\lhd} \leftarrow 0$; $E^{\rhd} \leftarrow 0$; $s \leftarrow 0$; $r \leftarrow 0$
04 $PK[\cdot] \leftarrow \bot$; $PK[0] \leftarrow pk$
05 $SK[\cdot] \leftarrow \bot$; $SK[0] \leftarrow sk$
06 $L_A[\cdot] \leftarrow \bot$; $L_B[\cdot] \leftarrow \bot$; $L_A[0] \leftarrow \diamond$
07 $st_A \leftarrow (PK, E, s, L_A, vfk, k.c, k.m, t)$
08 $st_B \leftarrow (SK, E, r, L_B, sgk, k.c, k.m, t)$
09 Return (st_A, st_B)

Proc $\mathrm{snd}_A(st_A, ad)$
10 $(PK, E, s, L, vfk, k.c, k.m, t) \leftarrow st_A$
11 $k^* \leftarrow \epsilon$; $ck \leftarrow_{\$} \mathcal{K}$; $C \leftarrow E^{\rhd} \,\|\, ck$
12 For $e' \leftarrow E^{\lhd}$ to $E^{\rhd}$:
13 · $(k, c) \leftarrow_{\$} \mathrm{enc}(PK[e'])$
14 $k^* \xleftarrow{\shortparallel} k$; $C \xleftarrow{\shortparallel} c$
15 · $\tau \leftarrow \mathrm{tag}(k.m, ad \,\|\, C)$
16 · $C \xleftarrow{\shortparallel} \tau$; $t \xleftarrow{\shortparallel} \rhd \,\|\, ad \,\|\, C$
17 · $k.o \,\|\, k.c \,\|\, k.m \,\|\, sk \leftarrow \mathrm{H}(k.c, k^*, t)$
18 · $pk \leftarrow \mathrm{gen}_{\mathsf{K}}(sk)$
19 $PK[\ldots, (E^{\rhd} - 1)] \leftarrow \bot$; $PK[E^{\rhd}] \leftarrow pk$
20 $E^{\lhd} \leftarrow E^{\rhd}$; $s \leftarrow s + 1$; $L[s] \leftarrow ad \,\|\, C$
21 $st_A \leftarrow (PK, E, s, L, vfk, k.c, k.m, t)$
22 Return $(S, k.o, C)$

Proc $\mathrm{snd}_B(st_B, ad)$
23 $(SK, E, r, L, sgk, k.c, k.m, t) \leftarrow st_B$
24 $(sk^*, pk^*) \leftarrow_{\$} \mathrm{gen}_{\mathsf{K}}$
25 $(sgk^*, vfk^*) \leftarrow_{\$} \mathrm{gen}_{\mathsf{S}}$
26 $E^{\rhd} \leftarrow E^{\rhd} + 1$; $SK[E^{\rhd}] \leftarrow sk^*$
27 $ck \leftarrow_{\$} \mathcal{K}$; $C \leftarrow r \,\|\, pk^* \,\|\, vfk^* \,\|\, ck$
28 $\sigma \leftarrow_{\$} \mathrm{sgn}(sgk, ad \,\|\, C)$
29 $C \xleftarrow{\shortparallel} \sigma$; $L[E^{\rhd}] \leftarrow \lhd \,\|\, ad \,\|\, C$
30 $st_B \leftarrow (SK, E, r, L, sgk^*, k.c, k.m, t)$
31 Return (st_B, C)

Proc $\mathrm{rcv}_B(st_B, ad, C)$
32 $(SK, E, r, L, sgk, k.c, k.m, t) \leftarrow st_B$
33 $t^* \leftarrow ad \,\|\, C$; $C \,\|\, \tau \leftarrow C$
34 · Require $\mathrm{vfy}_{\mathsf{M}}(k.m, ad \,\|\, C, \tau)$
35 $k^* \leftarrow \epsilon$; $e \,\|\, ck \,\|\, C \leftarrow C$
36 Require $E^{\lhd} \le e \le E^{\rhd}$
37 $t \xleftarrow{\shortparallel} L[E^{\lhd} + 1] \,\|\, \ldots \,\|\, L[e]$
38 $L[\ldots, e] \leftarrow \bot$
39 For $e' \leftarrow E^{\lhd}$ to e:
40 $c \,\|\, C \leftarrow C$
41 · $k \leftarrow \mathrm{dec}(SK[e'], c)$
42 · Require $k \neq \bot$
43 $k^* \xleftarrow{\shortparallel} k$
44 $t \xleftarrow{\shortparallel} \rhd \,\|\, t^*$
45 · $k.o \,\|\, k.c \,\|\, k.m \,\|\, sk \leftarrow \mathrm{H}(k.c, k^*, t)$
46 $SK[\ldots, (e-1)] \leftarrow \bot$; $SK[e] \leftarrow sk$
47 For $e' \leftarrow e + 1$ to $E^{\rhd}$:
48 $SK[e'] \leftarrow \mathrm{up}(SK[e'], t^*)$
49 $E^{\lhd} \leftarrow e$; $r \leftarrow r + 1$
50 $st_B \leftarrow (SK, E, r, L, sgk, k.c, k.m, t)$
51 Return $(st_B, k.o)$

Proc $\mathrm{rcv}_A(st_A, ad, C)$
52 $(PK, E, s, L, vfk, k.c, k.m, t) \leftarrow st_A$
53 $t \xleftarrow{\shortparallel} \lhd \,\|\, ad \,\|\, C$; $C \,\|\, \sigma \leftarrow C$
54 Require $\mathrm{vfy}_{\mathsf{S}}(vfk, ad \,\|\, C, \sigma)$
55 $r \,\|\, pk^* \,\|\, vfk \,\|\, ck \leftarrow C$
56 Require $L[r] \neq \bot$
57 $L[\ldots, (r-1)] \leftarrow \bot$; $L[r] \leftarrow \diamond$
58 For $s' \leftarrow r + 1$ to s:
59 $pk^* \leftarrow \mathrm{up}(pk^*, L[s'])$
60 $E^{\rhd} \leftarrow E^{\rhd} + 1$; $PK[E^{\rhd}] \leftarrow pk^*$
61 $st_A \leftarrow (PK, E, s, L, vfk, k.c, k.m, t)$
62 Return st_A

Figure 3.10: SRKE construction from key-updatable KEM $\mathsf{K} = (\mathrm{gen}_{\mathsf{K}}, \mathrm{enc}, \mathrm{dec})$, MAC $\mathsf{M} = (\mathrm{tag}, \mathrm{vfy}_{\mathsf{M}})$, one-time signature $\mathsf{S} = (\mathrm{gen}_{\mathsf{S}}, \mathrm{sgn}, \mathrm{vfy}_{\mathsf{S}})$, and random oracle H. For simplicity, we denote MAC key space and the spaces of chaining and collision keys with symbol $\mathcal{K}$. Notation: Lines 06,57: Storing placeholder symbol $\diamond$ for irrelevant array entries. Line 37: If $E^{\lhd} = e$ then no value shall be concatenated to t. Line 40: The last loop iteration clears C. Lines 16,53,44,29: Labels $\rhd$ and $\lhd$ distinguish transcript fragments in the A-to-B from those in B-to-A direction. Code lines tagged with a '·' depict the URKE construction's core.

Theorem 2 *The SRKE protocol* R *from Figure 3.10 offers key indistinguishability. More precisely, if function* H *is modeled as a random oracle, for every adversary* $\mathcal{A}$ *against SRKE scheme* R *in games* $\mathrm{KIND}^b_{\mathsf{R}}$ *from Figure 3.9 there exists an adversary* $\mathcal{B}_{\mathsf{K}}$ *against kuKEM* K *in game* KUOW *from Figure 3.3, an adversary* $\mathcal{B}_{\mathsf{S}}$ *against signature scheme* S *in game* SUF *from Figure 2.3, and an adversary* $\mathcal{B}_{\mathsf{M}}$ *against MAC* M *in game* SUF *from Figure 2.2 such that* $\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{R}}(\mathcal{A}) \leq 3 \cdot \mathrm{Adv}^{\mathrm{kuow}}_{\mathsf{K}}(\mathcal{B}_{\mathsf{K}}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{S}}(\mathcal{B}_{\mathsf{S}}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}}) + \frac{q_{\mathrm{H}}+2}{|\mathcal{K}|}$*, where* $\mathcal{K}$ *is the (collision-)key space, the running time of* $\mathcal{B}_{\mathsf{K}}$ *is about that of* $\mathcal{A}$ *plus* q_{H} *key checking and solve operations, the running time of* $\mathcal{B}_{\mathsf{S}}$ *and* $\mathcal{B}_{\mathsf{M}}$ *is about that of* $\mathcal{A}$*, and* q_{H} *is the number of* $\mathcal{A}$*'s random oracle queries.*

The proof of Theorem 2 is in Section 3.11. The approach is the same as in our URKE proof but with small yet important differences: 1) the proof reduces signature forgeries to the SUF security of the signature scheme to show that communication from B to A is authentic, 2) the security of session keys established by A is reduced to the KUOW security of the kuKEM. The reduction to the KUOW game is split into three cases: a) session keys established by A in sync, b) the first session key established by A out of sync, and c) all remaining session keys established by A out of sync. This distinction is made as in each of these cases a different encapsulated key—as part of the random oracle input—is shown to be unknown to the adversary. Finally the SRKE proof—as in the URKE proof—makes use of the MAC's SUF security to show that B will never establish challengeable keys out of sync.

3.7 Rationales for SRKE Design

We sketched the reasons for employing sophisticated primitives as building blocks for our design of SRKE in the previous section already. In this section we develop more detailed arguments for our design choices by providing attacks on constructions different from our design. At first it is described why SRKE requires signatures for protecting the communication from B to A—in contrast to employing

a MAC from A to B. Then we will evaluate the requirements for the KEM key pair update in the setting of concurrent sending of A and B.

3.7.1 Signatures from A to B

While a MAC suffices to protect authenticity for ciphertexts sent from A to B it does not suffice to protect the authenticity in the counter direction. The reason for this lies within the conditions with which future session keys of A and B are marked known in the KIND game of SRKE. An impersonation of A towards B has the same effect on the traceability of B's future session keys as if the adversary exposes B's state and then brings B out of sync. Either way all future session keys of B are marked known (see Figure 3.9 lines 37 and 54,38). In the first scenario, the adversary can compute the same session keys as B because the adversary initiates the key establishment impersonating A. In the second scenario, the adversary can comprehend B's computations during the receipt of ciphertexts because it possesses the same state information as B.

For computations of A, however, only the former scenario is applicable: if the adversary impersonated B towards A, then again the adversary is in the position to trace the establishment of session keys of A because it can simulate the respective counterpart's receiver computations. In contrast to this, when exposing A and then bringing her out of sync, according to the KIND game, the adversary must not obtain information on her future session keys (see Figure 3.9 lines 52 et seqq.). As a result, the exposure of A's state should not enable the adversary to impersonate B towards A. Consequently the authentication of the communication from B to A cannot be reached by a primitive with a symmetric secret but rather the protocol needs to ensure that B needs to be exposed in order to impersonate him towards A.

The non-trivial attack that is defended by employing signatures consists of the following adversary behavior: $st_A \leftarrow$ ExposeA; Extract authentication secrets from st_A to derive st'_B; $(C', st''_B) \leftarrow_{\$} \mathrm{snd}_B(st'_B, \epsilon)$; RcvA$(C', \epsilon)$;

$C_{A1} \leftarrow \text{SndA}(\epsilon)$; $k \leftarrow \text{Challenge}(A, 1)$. Thereby the adversary must not be able to decide whether it obtained the real or random key for ciphertext C_{A1} from the challenge oracle. Please note that this is related to *key-compromise impersonation* resilience (while in this case ephemeral signing keys are compromised).

3.7.2 Key-updatable KEM for Concurrent Sending

There exist two crucial properties that are required from the key pair update of the KEM in the setting in which A and B send concurrently. Firstly, the key update needs to be forward secure which means that an updated secret key does not reveal information on encapsulations to previous secret keys or to differently updated secret keys. Secondly, the update of the public key must not reveal information on keys that will be encapsulated to its respective secret key. We will explain the necessity of these requirements one after another.

The key pair update for concurrently sending only affects epochs that have been proposed by B, but that have not been processed by A yet. These updates have to consider ciphertexts that A sent during the transmission of the public key for a new epoch from B to A. Subsequently we describe an example scenario in which these updates are necessary for defending a non-trivial attack: In the worst case, all secrets among A and B have been exposed to the adversary before B proposes a new epoch ($st_A \leftarrow \text{ExposeA}$; $st_B \leftarrow \text{ExposeB}$). Thereby only a public key sent by B after the exposure will provide security for future session key establishments initiated by A. Now consider a scenario in which B proposes this new public key to A ($C_{B1} \leftarrow \text{SndB}(\epsilon)$; $\text{RcvA}(C_{B1}, \epsilon)$) and A is simultaneously impersonated towards B ($(st'_A, k', C') \leftarrow_{\$} \text{snd}_A(st_A, \epsilon)$; $\text{RcvB}(C', \epsilon)$). Since B proposed the new public key within C_{B1} in sync and A received it in sync respectively—and B was not exposed under the new state—, future established session keys of A are considered to be indistinguishable from random key space elements again ($C_{A1} \leftarrow \text{SndA}(\epsilon)$; $k \leftarrow \text{Challenge}(A, 1)$). Due to the impersonation of A towards B, however, B became out of sync. Becoming out of sync cannot be detected by B because

the adversary can send a valid ciphertext C' under the exposed state of A st_A. Exposing B out of sync afterwards ($st'_B \leftarrow \text{ExposeB}$), by definition, must not have an impact on the security of session keys established by A afterwards (see Figure 3.9 line 55). As a result, after the adversary performed these steps, the challenged session key is required to remain indistinguishable from a random element from the key space. Consequently B must perform an update of the secret key for the newest epoch when receiving C' such that the public key transmitted in C_{B1} still provides its security guarantees when using it in A's final send operation (remember that all previous secrets among A and B were exposed before).

When accepting that an update of B's future epoch's secret keys is required at the receipt of ciphertexts, another condition arises for the respective update of A's public keys. For maintaining correctness, A of course needs to compute updates of a received new public key with respect to all previously sent ciphertexts that B was not aware of when sending the public key. Suppose again, in a fresh session, that A's and B's secrets have all been exposed towards the adversary ($st_A \leftarrow \text{ExposeA}$; $st_B \leftarrow \text{ExposeB}$). Now A sends a new key establishing ciphertext and B proposes a new epoch public key ($C_{A1} \leftarrow \text{SndA}(\epsilon)$; $C_{B1} \leftarrow \text{SndB}(\epsilon)$). According to the previous paragraph, A needs to update the received public key in C_{B1} with respect to C_{A1} after receiving C_{B1} ($\text{RcvA}(C_{B1}, \epsilon)$). Since C_{B1} introduces a new epoch, the next send operation of A needs to establish a secure session key again ($C_{A2} \leftarrow \text{SndA}(\epsilon)$; $k \leftarrow \text{Challenge}(A, 2)$). Now observe that in order to update the received public key, A can only use information from her state st_A—which is known by the adversary—, public information like the transmitted ciphertexts, and randomness. Essentially, the update can hence only depend on information that the adversary knows plus random coins which cannot be transmitted confidentially to B before performing the update (because there exist no secrets except for the key pair that first needs to be updated). Since B probably received C_{A1} before A received C_{B1}, A cannot influence the update performed by B on his secret key. This means that the updates of A and B need to be conducted independently. As such, the adversary is able

to perform the update on the same information that A has (only randomness of A and of the adversary can differ). Nevertheless, both updates—the one performed by the adversary and the one performed by A—need to be compatible to the secret key that B derives from his update. As a result, the update of the public key must not reveal the respective secret key (or any other information that can be used to obtain information on keys encapsulated to this updated public key). Otherwise, the adversary would obtain this information as well and thereby the security of the key established with C_{A2} would not be preserved.

Both requirements are reflected in the security of kuKEM (see Figure 3.3).

While these observations are only heuristic arguments for relying on kuKEM, our analysis in Chapter 4 provably reveals conditions under which kuKEM and RKE are equivalent such that the former is both necessary and sufficient to realize the latter.

3.7.3 Encapsulation to all Public Keys

In order to explain why A always encapsulates to all public keys in her state, we describe a scenario in which A only maintains one public key in her state to which she can securely encapsulate keys (while the state contains multiple *useless* public keys). This scenario is crucial because A does not know, which of her public keys provides security, and the SRKE protocol is required to output secure session keys therein. Consequently only encapsulating to all public keys in A's state solves the underlying issue. The reasons for encapsulating to all public keys in A's state is closely related to the reasons for employing a kuKEM in SRKE (see the previous Subsection).

Assume the adversary exposes the states of both parties ($st_A \leftarrow \text{ExposeA}$; $st_B \leftarrow \text{ExposeB}$). Consequently none of A's public keys provides any security guarantees for the encapsulation towards the adversary anymore. If the adversary lets B send a ciphertext and thereby propose a new public key to A, A's future session keys are required to be secure again ($C_{B1} \leftarrow \text{SndB}(\epsilon)$; $\text{RcvA}(C_{B1}, \epsilon)$). Impersonating A to-

wards B and then exposing B to obtain his state has—according to the KIND game—no influence on the traceability of A's future session keys $\big((st'_A, k', C') \leftarrow_{\$} \mathrm{snd}_A(st_A, \epsilon);\ \mathrm{RcvB}(C', \epsilon);\ st'_B \leftarrow \mathrm{ExposeB}\big)$. However, our construction allows the adversary to impersonate B towards A afterwards: the impersonation of A towards B only *invalidates* the kuKEM secret key in B's state via the key update in B's receive algorithm. The signing key in B's state is still valid for the communication to A since it was not modified at the receipt of the impersonating ciphertext. As such, the adversary may use the signing key and then implant further public keys in A's state by sending these public keys to A $\big((st''_B, C'') \leftarrow_{\$} \mathrm{snd}_B(st'_B, \epsilon);\ \mathrm{RcvA}(C'', \epsilon)\big)$. These public keys do not provide security with respect to A's session keys since the adversary can freely choose them. As a result, only the public key that B sent in sync before A was impersonated towards B belongs to a secret key that the adversary does not know (public key in C_{B1}). Since A has no indication which public key's secret key is not known by the adversary (note that A and B were exposed at the beginning of the presented scenario and the adversary planted own public keys in A's state at the end of the scenario by sending valid ciphertexts), A needs to encapsulate to all public keys in order to obtain at least one encapsulated key as secret input to the random oracle such that the session key also remains secure $\big(C_{A1} \leftarrow \mathrm{SndA}(\epsilon);\ k \leftarrow \mathrm{Challenge}(A, 1)\big)$.

Observe that the scenario, described above, lacks an argument why also the first public key in A's state needs to be used for the encapsulation if A received further public keys from B afterwards. The reason for also using the first public key, that is always derived from the previous random oracle output, lies within A's sending after becoming out of sync. A became out of sync by receiving C'' (see above). When sending C_{A1}, A derived a new public key for her state. The secret key to this public key was part of the same random oracle output as the session key that is challenged afterwards (and established with C_{A1}). As argued before, this session key is secure (for all details we refer the reader to the proof in Section 3.11). Consequently the public key in A's state after sending C_{A1} provides security against the adversary regrading encapsulations. However, the adversary can still plant new

public keys to A's state $((st'''_B, C''') \leftarrow_{\$} \mathrm{snd}_B(st''_B, \epsilon); \mathrm{RcvA}(C''', \epsilon))$. As such, only the first public key in A's state provides security after A became out of sync (and sent once afterwards). All remaining public keys may belong to secret keys chosen by the adversary. Since A will not notice when she became out of sync, she also needs to include the first public key in her state for encapsulating within her send algorithm in order to compute secure session keys $(C_{A2} \leftarrow \mathrm{SndA}(\epsilon);\ k \leftarrow \mathrm{Challenge}(A, 2))$.

As a result, A always needs to encapsulate to all public keys in her state such that at least one encapsulated key is a secret input to the random oracle (in case her future session keys were not marked traceable by the KIND game).

3.8 Bidirectionally ratcheted key exchange (BRKE)

The URKE and SRKE primitives are unbalanced in that they allow only one of the two participants to actively establish new keys. As the ratcheting notion first appeared in the context of (bidirectional) instant messaging [Lan16, BGB04, PM16] it is natural to ask for a fully balanced primitive where both participants have the capability of establishing fresh keys independently of each other. In this section we correspondingly study *bidirectional ratcheted key exchange* (BRKE) by first defining its syntax, functionality, and security, and then proposing two constructions. We note that the BRKE notions are natural extensions of those of URKE and SRKE, effectively duplicating specific parts of the security model and constructions so that they are available in both directions of communication. The main challenge is to properly interweave the communication in the two directions.

Formally, a BRKE scheme for a finite key space $\mathcal{K}$ and an associated-data space $\mathcal{AD}$ is a triple $\mathsf{R} = (\mathrm{init}, \mathrm{snd}, \mathrm{rcv})$ of algorithms together with a state space $\mathcal{S}$ and a ciphertext space $\mathcal{C}$. The randomized initialization algorithm init returns a pair of states $(st_A, st_B) \in \mathcal{S} \times \mathcal{S}$. The randomized sending algorithm snd takes a state $S \in \mathcal{S}$ and

an associated-data string $ad \in \mathcal{AD}$, and produces an updated state $S' \in \mathcal{S}$, a key $k \in \mathcal{K}$, and a ciphertext $c \in \mathcal{C}$. Finally, the deterministic receiving algorithm rcv takes a state $S \in \mathcal{S}$, an associated-data string $ad \in \mathcal{AD}$, and a ciphertext $c \in \mathcal{C}$, and *either* outputs an updated state $S' \in \mathcal{S}$ and a key $k \in \mathcal{K}$ *or* outputs the special symbol $\perp$ to indicate rejection. A shortcut notation for these syntactical definitions (conceptually shown in Figure 3.1) is

$$\begin{array}{rcccl} & & \text{init} & \to_{\$} & \mathcal{S} \times \mathcal{S} \\ \mathcal{S} \times \mathcal{AD} & \to & \text{snd} & \to_{\$} & \mathcal{S} \times \mathcal{K} \times \mathcal{C} \\ \mathcal{S} \times \mathcal{AD} \times \mathcal{C} & \to & \text{rcv} & \to & (\mathcal{S} \times \mathcal{K}) \cup \{(\perp, \perp)\} \ . \end{array}$$

Correctness of BRKE. We formalize the correctness of BRKE via game FUNC in Figure 3.11. Concretely, we say scheme R is *correct* if $\Pr[\text{FUNC}_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1] = 0$ for all adversaries $\mathcal{A}$. The game is best understood by comparing it with the functionality game of SRKE (in Figure 3.8): As in BRKE the roles of the participants are symmetric, the Snd and Rcv oracles in Figure 3.11 are effectively the amalgamation of the SndA and SndB oracles, respectively, the RcvA and RcvB oracles, from Figure 3.8. Observe that, as in BRKE the snd invocations of both participants create fresh keys and start new epochs, in the FUNC game each participant has its individual copy of the game variables $\text{K}, \text{EP}, e, E^{\lhd}, E^{\rhd}$; this is in contrast with the SRKE case where variables K, EP, e were specific to one party, and variables $E^{\lhd}, E^{\rhd}$ were specific to the other.

Security of BRKE. Our BRKE security model is derived by lifting the indistinguishability notion from SRKE from Figure 3.9 to the fully bidirectional case, again amalgamating SndA and SndB oracles and RcvA and RcvB oracles to single Snd and Rcv oracles, respectively, and using the notation of the BRKE functionality game from Figure 3.11. The result are the key indistinguishability games $\text{KIND}^b_{\mathsf{R}}$, for challenge bit $b \in \{0, 1\}$, specified in Figure 3.12. The only noteable novelty, required as in BRKE keys can be established by both participants, is that the game manages two copies of the K array per user:

Game $\mathrm{FUNC}_{\mathsf{R}}(\mathcal{A})$
00 For $u \in \{A, B\}$:
01 $s_u \leftarrow 0$; $r_u \leftarrow 0$
02 $e_u \leftarrow 0$; $\mathrm{EP}_u[\cdot] \leftarrow \bot$
03 $E_u^{\leqslant} \leftarrow 0$; $E_u^{>|} \leftarrow 0$
04 $\mathrm{AC}_u[\cdot] \leftarrow \bot$; $is_u \leftarrow \mathtt{T}$
05 $\mathrm{K}_u[\cdot] \leftarrow \bot$
06 $(st_A, st_B) \leftarrow_{\$} \mathrm{init}$
07 Invoke $\mathcal{A}$
08 Stop with 0

Oracle $\mathrm{Snd}(u, ad)$
09 Require $st_u \neq \bot$
10 $(st_u, k, c) \leftarrow_{\$} \mathrm{snd}(st_u, ad)$
11 If is_u:
12 $\mathrm{AC}_u[s_u] \leftarrow (ad, c)$
13 $\mathrm{EP}_u[s_u] \leftarrow e_u$
14 $E_u^{>|} \leftarrow E_u^{>|} + 1$
15 $\mathrm{K}_u[s_u] \leftarrow k$
16 $s_u \leftarrow s_u + 1$
17 Return c

Oracle $\mathrm{Rcv}(u, ad, c)$
18 Require $st_u \neq \bot$
19 If $is_u \wedge \mathrm{AC}_{\bar{u}}[r_u] \neq (ad, c)$:
20 $is_u \leftarrow \mathtt{F}$
21 If is_u:
22 $E_u^{\leqslant} \leftarrow \mathrm{EP}_{\bar{u}}[r_u]$
23 $e_u \leftarrow e_u + 1$
24 $(st_u, k) \leftarrow \mathrm{rcv}(st_u, ad, c)$
25 If $st_u = \bot$: Return $\bot$
26 Reward $is_u \wedge k \neq \mathrm{K}_{\bar{u}}[r_u]$
27 $r_u \leftarrow r_u + 1$
28 Return

Figure 3.11: Game FUNC for BRKE scheme R. For a user $u \in \{A, B\}$ we write $\bar{u}$ for its peer; that is, we always have $\{u, \bar{u}\} = \{A, B\}$.

We represent keys that user u establishes with the role of a sender as $\mathrm{K}_u[\mathtt{S}, \ldots]$, and we represent keys that u recovers as a receiver as $\mathrm{K}_u[\mathtt{R}, \ldots]$. For a BRKE scheme R, with any adversary $\mathcal{A}$ we associate its key distinguishing advantage $\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{R}}(\mathcal{A}) := |\Pr[\mathrm{KIND}^1_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathrm{KIND}^0_{\mathsf{R}}(\mathcal{A}) \Rightarrow 1]|$. Intuitively, R offers key indistinguishability if all practical adversaries have a negligible key distinguishing advantage.

3.9 Constructing BRKE

We propose two constructions of the BRKE primitive. Their common denominator is that they internally use two instances of a SRKE protocol—one in the Alice-to-Bob direction and one in the Bob-to-Alice direction. The challenge is to properly interweave their oper-

Game $\mathrm{KIND}^b_{\mathsf{R}}(\mathcal{A})$
00 For $u \in \{A, B\}$:
01 $s_u \leftarrow 0$; $r_u \leftarrow 0$
02 $e_u \leftarrow 0$; $\mathrm{EP}_u[\cdot] \leftarrow \bot$
03 $E^{\lhd}_u \leftarrow 0$; $E^{\rhd}_u \leftarrow 0$
04 $\mathrm{AC}_u[\cdot] \leftarrow \bot$; $is_u \leftarrow \mathtt{T}$
05 $\mathrm{K}_u[\cdot] \leftarrow \bot$; $\mathrm{XP}_u \leftarrow \emptyset$
06 $\mathrm{KN}_u \leftarrow \emptyset$; $\mathrm{CH}_u \leftarrow \emptyset$
07 $(st_A, st_B) \leftarrow_{\$} \mathrm{init}$
08 $b' \leftarrow_{\$} \mathcal{A}$
09 For $u \in \{A, B\}$:
10 Require $\mathrm{KN}_u \cap \mathrm{CH}_u = \emptyset$
11 Stop with b'

Oracle $\mathrm{Snd}(u, ad)$
12 Require $st_u \neq \bot$
13 $(st_u, k, c) \leftarrow_{\$} \mathrm{snd}(st_u, ad)$
14 If is_u:
15 $\mathrm{AC}_u[s_u] \leftarrow (ad, c)$
16 $\mathrm{EP}_u[s_u] \leftarrow e_u$
17 $E^{\rhd}_u \leftarrow E^{\rhd}_u + 1$
18 $\mathrm{K}_u[\mathtt{S}, e_u, s_u] \leftarrow k$
19 $s_u \leftarrow s_u + 1$
20 Return c

Oracle $\mathrm{Reveal}(u, i)$
as in URKE/SRKE (Fig. 3.6)

Oracle $\mathrm{Rcv}(u, ad, c)$
21 Require $st_u \neq \bot$
22 If $is_u \wedge \mathrm{AC}_{\bar{u}}[r_u] \neq (ad, c)$:
23 $is_u \leftarrow \mathtt{F}$
24 If $r_u \in \mathrm{XP}_{\bar{u}}$:
25 $\mathrm{KN}_u \xleftarrow{\cup} \{\mathtt{S}\} \times \mathbb{N} \times [\, s_u, \ldots]$
26 $\mathrm{KN}_u \xleftarrow{\cup} \{\mathtt{R}\} \times \mathbb{N} \times [\, r_u, \ldots]$
27 If is_u:
28 $E^{\lhd}_u \leftarrow \mathrm{EP}_{\bar{u}}[r_u]$
29 $e_u \leftarrow e_u + 1$
30 $(st_u, k) \leftarrow \mathrm{rcv}(st_u, ad, c)$
31 If $st_u = \bot$: Return $\bot$
32 If is_u: $k \leftarrow \diamond$
33 $\mathrm{K}_u[\mathtt{R}, E^{\lhd}_u, r_u] \leftarrow k$
34 $r_u \leftarrow r_u + 1$
35 Return

Oracle $\mathrm{Expose}(u)$
36 $\mathrm{KN}_u \xleftarrow{\cup} \{\mathtt{R}\} \times [E^{\lhd}_u \,..\, E^{\rhd}_u] \times [\, r_u, \ldots]$
37 If is_u:
38 $\mathrm{XP}_u \xleftarrow{\cup} \{s_u\}$
39 $\mathrm{KN}_{\bar{u}} \xleftarrow{\cup} \{\mathtt{S}\} \times [E^{\lhd}_u \,..\, E^{\rhd}_u] \times [\, r_u, \ldots]$
40 Return st_u

Oracle $\mathrm{Challenge}(u, i)$
as in URKE/SRKE (Fig. 3.6)

Figure 3.12: Games KIND^b, $b \in \{0, 1\}$, for BRKE scheme R. Symbols $\mathtt{S}$ and $\mathtt{R}$ are labels that distinguish whether keys were established in a sending or a receiving operation.

ations such that, overall, BRKE security is reached. (For instance, attacking one of the instances needs to automatically escalate to an attack on the second as otherwise attacks on KIND security become feasible). Our first solution achieves this via strongly unforgeable one-time signatures. Our second solution is slightly more efficient but ad-hoc; it is derived from the specific SRKE protocol from Figure 3.10 and carefully interleaves the use of its inner building blocks.

Generic construction with one-time signatures. Let SR de-

note a SRKE protocol, and assume a strongly unforgeable one-time signature scheme as an auxiliary building block. The snd and rcv algorithms of our first BRKE construction are in Figure 3.13. (The init algorithm is not in the figure; it just performs two invocations of $\text{init}_{\mathsf{SR}}$, and the initial states of users consist of one sending and one receiving state.) Concretely, our snd algorithm performs internally two snd invocations of the underlying SRKE scheme (in lines 02,03), which results in a key $k.o$ and two ciphertexts c_1, c_2. These ciphertexts are protected by a one-time signature before being sent to the peer: a fresh signature key pair is generated per snd invocation (in line 01), and the pair c_1, c_2 signed with it (in line 04). Note that the signature verification key is included in the associated-data field of both internal snd invocations (see line 01). To allow for signature verification on the side of the peer, the verification key is sent along with the ciphertexts. The peer processes the ciphertext in the obvious way.

We describe the rationale behind our construction. The goal is to bind the two ciphertext components c_1, c_2 together such that any manipulation of the pair will be detected by both underlying SRKE instances. One could try to implement this directly via the associated-data fields on the SRKE, that is, by including c_1 in ad when producing c_2 or by including c_2 in ad when producing c_1. It turns out that both these options are too weak and allow for attacks on key indistinguishability of the composed BRKE scheme. By using the one-time signature scheme we side-step this one-before-the-other dependency. The security argument for our construction in Figure 3.13 is as follows: Note first that each verification key recovered in line 09 is either authentic or not. If it is, then also c_1, c_2, σ are authentic (otherwise the adversary would have broken the strong unforgeability of the one-time signature scheme). If it is not, then this will be reflected in the changed associated-data field ad line 09, i.e., both SRKE instances will be notified of this.

Theorem 3 *The BRKE protocol* BR *from Figure 3.13 offers key indistinguishability. More precisely, for every adversary* $\mathcal{A}$ *against BRKE scheme* BR *in games* $\text{KIND}^b_{\mathsf{BR}}$ *from Figure 3.12 there exists an adver-*

Proc $\mathrm{snd}(st, ad)$	**Proc** $\mathrm{rcv}(st, ad, c)$
00 $(st_1, st_2) \leftarrow st$	08 $(st_1, st_2) \leftarrow st$
01 $(sgk, vfk) \leftarrow_{\$} \mathrm{gen}_{\mathsf{S}}$; $ad \xleftarrow{\shortparallel} vfk$	09 $vfk \,\|\, c_1 \,\|\, c_2 \,\|\, \sigma \leftarrow c$; $ad \xleftarrow{\shortparallel} vfk$
02 $(st_1, k.o, c_1) \leftarrow_{\$} \mathrm{snd}_A(st_1, ad)$	10 Require $\mathrm{vfy}_{\mathsf{S}}(vfk, c_1 \,\|\, c_2, \sigma)$
03 $(st_2, c_2) \leftarrow_{\$} \mathrm{snd}_B(st_2, ad)$	11 $st_1 \leftarrow \mathrm{rcv}_A(st_1, ad, c_2)$
04 $\sigma \leftarrow_{\$} \mathrm{sgn}(sgk, c_1 \,\|\, c_2)$	12 Require $st_1 \neq \bot$
05 $c \leftarrow vfk \,\|\, c_1 \,\|\, c_2 \,\|\, \sigma$	13 $(st_2, k.o) \leftarrow \mathrm{rcv}_B(st_2, ad, c_1)$
06 $st \leftarrow (st_1, st_2)$	14 Require $st_2 \neq \bot$
07 Return $(st, k.o, c)$	15 $st \leftarrow (st_1, st_2)$
	16 Return $(st, k.o)$

Figure 3.13: Generic construction of BRKE scheme BR from a SRKE scheme $\mathsf{SR} = (\mathrm{init}_{\mathsf{SR}}, \mathrm{snd}_A, \mathrm{snd}_B, \mathrm{rcv}_A, \mathrm{rcv}_B)$ and a one-time signature scheme $\mathsf{S} = (\mathrm{gen}_{\mathsf{S}}, \mathrm{sgn}, \mathrm{vfy}_{\mathsf{S}})$.

sary $\mathcal{B}_{\mathsf{SR}}$ *against SRKE scheme* SR *in game* $\mathrm{KIND}_{\mathsf{SR}}$ *from Figure 3.9 and an adversary* $\mathcal{B}_{\mathsf{S}}$ *against signature scheme* S *in game* SUF *from Figure 2.3 such that* $\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{BR}}(\mathcal{A}) \leq 2\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{SR}}(\mathcal{B}_{\mathsf{SR}}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{S}}(\mathcal{B}_{\mathsf{S}})$*, where the running times of* $\mathcal{B}_{\mathsf{SR}}$ *and* $\mathcal{B}_{\mathsf{S}}$ *are about that of* $\mathcal{A}$*.*

We prove Theorem 3 in Section 3.12.

Ad-hoc construction. Our ad-hoc construction in Figure 3.14 directly adopts the SRKE construction and combines both sending algorithms and both receiving algorithms. To derive the necessary binding that was described in the previous paragraph, the sending algorithms intertwine by signing both ciphertext parts together and then feeding the whole ciphertext—including the signature—into the random oracle. As such, a manipulation of parts of the ciphertext directly affects both SRKE states.

We split the blocks taken from a different algorithm of the SRKE construction respectively by leaving blank lines in Figure 3.14. All lines not marked with a '·' at the left margin are directly copied from the SRKE construction. In line 20 instead of setting the ciphertext to the newest epoch number, the ciphertext is appended by this number. As a result, both ciphertexts of the sending algorithms of SRKE are concatenated. In line 25 we index the encoding by the user identifier of the sending party. While in SRKE, each algorithm can only be used by

one of the two communicating parties, in BRKE a unique encoding for each party becomes necessary to separate the parts of the transcript with respect to their origin. Similarly the encoding in the receive algorithm is equipped with user indexed encoding (see lines 37,56). In order to input the whole just received ciphertext and associated-data string to the random oracle, in line 37 a copy of it is stored in t^* (in line 56 this string is appended to the current transcript). Finally in line 39 the ciphertexts of both SRKE instantiations are split again to process them at receipt.

Please note that the whole ciphertext is thereby signed at sending and fed into the random oracle. As such, the ciphertexts of SRKE are authenticated in this ad-hoc BRKE construction without employing an additional one-time signature.

Proc init
00 For $u \in \{A, B\}$:
01 $(sgk_u, vfk_u) \leftarrow_{\$} \mathrm{gen}_{\mathsf{S}}$
02 $(sk_u, pk_u) \leftarrow_{\$} \mathrm{gen}_{\mathsf{K}}$
03 $k.c_u \leftarrow_{\$} \mathcal{K}$; $t \leftarrow \epsilon$
04 $E^{\leqslant} \leftarrow 0$; $E^{\rangle} \leftarrow 0$
05 $s \leftarrow 0$; $r \leftarrow 0$
06 $PK_u[\cdot] \leftarrow \perp$; $PK_u[0] \leftarrow pk$
07 $SK_u[\cdot] \leftarrow \perp$; $SK_u[0] \leftarrow sk$
08 $L_S[\cdot] \leftarrow \perp$; $L_R[\cdot] \leftarrow \perp$; $L_S[0] \leftarrow \diamond$
09 $S_u \leftarrow (PK_{\bar{u}}, E, s, L_S, vfk_{\bar{u}}, k.c_{\bar{u}}, t)$
10 $R_u \leftarrow (SK_u, E, r, L_R, sgk_u, k.c_u, t)$
11 $ST_u \leftarrow (R_u, S_u)$
12 Return (ST_A, ST_B)

Proc snd(ST, ad)
13 $(R, S) \leftarrow ST$
14 $(SK, E_R, r, L_R, sgk, k.c_R, t_R) \leftarrow R$
15 $(sk^*, pk^*) \leftarrow_{\$} \mathrm{gen}_{\mathsf{K}}$
16 $(sgk^*, vfk^*) \leftarrow_{\$} \mathrm{gen}_{\mathsf{S}}$
17 $E_R^{\rangle} \leftarrow E_R^{\rangle} + 1$; $SK[E_R^{\rangle}] \leftarrow sk^*$
18 $ck \leftarrow_{\$} \mathcal{K}$; $C \leftarrow r \,\|\, pk^* \,\|\, vfk^* \,\|\, ck$

19 $(PK, E_S, s, L_S, vfk, k.c_S, t_S) \leftarrow S$
20 · $k^* \leftarrow \epsilon$; $C \xleftarrow{\shortparallel} E_S^{\rangle}$
21 For $e' \leftarrow E_S^{\leqslant}$ to $E_S^{\rangle}$:
22 $(k, c) \leftarrow_{\$} \mathrm{enc}(PK[e'])$
23 $k^* \xleftarrow{\shortparallel} k$; $C \xleftarrow{\shortparallel} c$

24 $\sigma \leftarrow_{\$} \mathrm{sgn}(sgk, ad \,\|\, C)$
25 · $C \xleftarrow{\shortparallel} \sigma$; $L_R[E_R^{\rangle}] \leftarrow \triangleright_u \,\|\, ad \,\|\, C$
26 $R \leftarrow (SK, E_R, r, L_R, sgk^*, k.c_R, t_R)$

27 $t_S \xleftarrow{\shortparallel} \triangleright_u \,\|\, ad \,\|\, C$
28 $k.o \,\|\, k.c_S \,\|\, sk \leftarrow \mathrm{H}(k.c_S, k^*, t_S)$
29 $pk \leftarrow \mathrm{gen}_{\mathsf{K}}(sk)$
30 $PK[\ldots, (E_S^{\rangle} - 1)] \leftarrow \perp$; $PK[E_S^{\rangle}] \leftarrow pk$
31 $E_S^{\leqslant} \leftarrow E_S^{\rangle}$; $s \leftarrow s + 1$; $L_S[s] \leftarrow ad \,\|\, C$
32 $S \leftarrow (PK, E_S, s, L_S, vfk, k.c_S, t_S)$
33 $ST \leftarrow (R, S)$
34 Return $(ST, k.o, C)$

Proc rcv(ST, ad, C)
35 $(R, S) \leftarrow ST$
36 $(PK, E_S, s, L_S, vfk, k.c_S, t_S) \leftarrow S$
37 · $t^* \leftarrow ad \,\|\, C$; $t_S \xleftarrow{\shortparallel} \triangleright_{\bar{u}} \,\|\, t^*$; $C \,\|\, \sigma \leftarrow C$
38 Require $\mathrm{vfy}_{\mathsf{S}}(vfk, ad \,\|\, C, \sigma)$
39 · $r \,\|\, pk^* \,\|\, vfk \,\|\, ck \,\|\, C \leftarrow C$
40 Require $L_S[r] \neq \perp$
41 $L_S[\ldots, (r-1)] \leftarrow \perp$; $L_S[r] \leftarrow \diamond$
42 For $s' \leftarrow r + 1$ to s:
43 $pk^* \leftarrow \mathrm{up}(pk^*, L_S[s'])$
44 $E_S^{\rangle} \leftarrow E_S^{\rangle} + 1$; $PK[E_S^{\rangle}] \leftarrow pk^*$
45 $S \leftarrow (PK, E_S, s, L_S, vfk, k.c_S, t_S)$

46 $(SK, E_R, r, L_R, sgk, k.c_R, t_R) \leftarrow R$
47 $k^* \leftarrow \epsilon$; $e \,\|\, C \leftarrow C$
48 Require $E_R^{\leqslant} \leq e \leq E_R^{\rangle}$
49 $t_R \xleftarrow{\shortparallel} L_R[E_R^{\leqslant} + 1] \,\|\, \ldots \,\|\, L_R[e]$
50 $L_R[\ldots, e] \leftarrow \perp$
51 For $e' \leftarrow E_R^{\leqslant}$ to e:
52 $c \,\|\, C \leftarrow C$
53 $k \leftarrow \mathrm{dec}(SK[e'], c)$
54 Require $k \neq \perp$
55 $k^* \xleftarrow{\shortparallel} k$
56 · $t_R \xleftarrow{\shortparallel} \triangleright_{\bar{u}} \,\|\, t^*$
57 $k.o \,\|\, k.c_R \,\|\, sk \leftarrow \mathrm{H}(k.c_R, k^*, t_R)$
58 $SK[\ldots, (e-1)] \leftarrow \perp$; $SK[e] \leftarrow sk$
59 For $e' \leftarrow e + 1$ to $E_R^{\rangle}$:
60 $SK[e'] \leftarrow \mathrm{up}(SK[e'], t^*)$
61 $E_R^{\leqslant} \leftarrow e$; $r \leftarrow r + 1$
62 $R \leftarrow (SK, E_R, r, L_R, sgk, k.c_R, t_R)$
63 $ST \leftarrow (R, S)$
64 Return $(ST, k.o)$

Figure 3.14: Construction of BRKE from our SRKE construction in Figure 3.10 by intertwining the respective algorithms.

3.10 Proof of URKE

Overview

In order to prove the construction's security, we proceed in a sequence of games that pursues two targets: on the one hand the simulation is to be conducted without the usage of secret keys that belong to A's public keys, and on the other hand the adversary's random oracle requests and its forged MAC tags are used to solve the underlying hardness assumptions. While games G_2, G_3 and G_4 answer the former purpose, G_3 and G_5 (and in SRKE G_1) entail abortions of the game for events that are assumed to occur with negligible probability.

To unify the proof description for URKE and SRKE, we harmonize the numbering of the games. As we consider signature forgeries in SRKE in game G_1 and the URKE construction makes no use of signatures, G_1 in URKE equals the original game and is thereby a placeholder.

The description of the game hops for the SRKE proof consequently mainly focuses on the differences and peculiarities.

Notation Figure 3.15 and 3.16 show the proof, split into the KIND game, containing the URKE construction, and the random oracle. Modifications by the game hops are included into the figures and denoted as follows:

The symbol at the right margin of a line annotates for which games a manipulation due to the game hop is valid. A line marked with a symbol $G_{\geq 3}$ is valid for game G_3 and all subsequent games. If a line is not valid for the final game, this line is struck through. Thereby either only the game is denoted in which a line becomes invalid by $G_{<y}$ or an interval of games for which a line is valid is marked by G_{x-y} where G_x is the game in which a line is introduced and G_y is the first game in which the line is disregarded.

Procedures, newly introduced by a game hop, are denoted by the symbol described above only in the first line of the procedure to reduce redundancy (e.g., see the procedures in Figure 3.16).

Simulation $\mathcal{S}^b_{\mathsf{R}}(\mathcal{A})$

00 $\mathrm{R}[\cdot] \leftarrow \bot$ $\quad \mathbf{G_{\geq 0}}$
01 $s_A \leftarrow 0$; $r_B \leftarrow 0$
02 $\mathrm{AC}_A[\cdot] \leftarrow \bot$; $is_B \leftarrow \mathtt{T}$
03 $oos_B \leftarrow \infty$; $acos_B \leftarrow \bot$ $\quad \mathbf{G_{\geq 3}}$
04 $\mathrm{K}_A[\cdot] \leftarrow \bot$; $\mathrm{K}_B[\cdot] \leftarrow \bot$
05 $\mathrm{XP}_A \leftarrow \emptyset$
06 $\mathrm{KN}_A \leftarrow \emptyset$; $\mathrm{KN}_B \leftarrow \emptyset$
07 $\mathrm{CH}_A \leftarrow \emptyset$; $\mathrm{CH}_B \leftarrow \emptyset$
08 $SK_\star[\cdot] \leftarrow \bot$; $KT_\star[\cdot] \leftarrow \bot$ $\quad \mathbf{G_{\geq 2}}$
09 $CK[\cdot] \leftarrow \bot$; $XSK \leftarrow \emptyset$ $\quad \mathbf{G_{\geq 2}}$
10 $sk \leftarrow_{\$} \mathcal{SK}$
11 $pk \leftarrow \mathrm{gen}_{\mathsf{K}}(sk)$
12 $(k.c, k.m) \leftarrow_{\$} \mathcal{K}^2$; $t \leftarrow \epsilon$
13 $st_A \leftarrow (pk, k.c, k.m, t)$
14 ~~$st_B \leftarrow (sk, k.c, k.m, t)$~~ $\quad \mathbf{G_{<2}}$
15 $KT_\star[s_A, \mathtt{S}] \leftarrow (k.c, k.m, t)$ $\quad \mathbf{G_{\geq 2}}$
16 $SK_\star[s_A, \mathtt{S}] \leftarrow sk$ $\quad \mathbf{G_{\geq 2}}$
17 $b' \leftarrow_{\$} \mathcal{A}$
18 Require $\mathrm{KN}_A \cap \mathrm{CH}_A = \emptyset$
19 Require $\mathrm{KN}_B \cap \mathrm{CH}_B = \emptyset$
20 Stop with b'

Oracle SndA(ad)

21 $i \leftarrow (s_A, \mathtt{S})$ $\quad \mathbf{G_{\geq 2}}$
22 $(pk, k.c, k.m, t) \leftarrow st_A$
23 $(k, c) \leftarrow_{\$} \mathrm{enc}(pk)$
24 $CK[c, i] \leftarrow k$ $\quad \mathbf{G_{\geq 2}}$
25 $ck \leftarrow_{\$} \mathcal{K}$
26 $\tau \leftarrow \mathrm{tag}(k.m, ad \,\|\, c \,\|\, ck)$
27 $C \leftarrow c \,\|\, ck \,\|\, \tau$
28 $t \xleftarrow{\cup} ad \,\|\, C$
29 ~~$k.o \,\|\, k.c^* \,\|\, k.m \,\|\, sk \leftarrow \mathrm{H}(k.c, k, t)$~~ $\quad \mathbf{G_{<3}}$
30 $k.o \,\|\, k.c^* \,\|\, \_ \leftarrow \mathrm{G}(k.c, t, i, \mathtt{T})$ $\quad \mathbf{G_{\geq 3}}$
31 $k.m \leftarrow_{\$} \mathcal{K}$; $sk \leftarrow_{\$} \mathcal{SK}$ $\quad \mathbf{G_{\geq 3}}$
32 ~~$\mathrm{SetO}(k.c, t, i, k.m, sk)$~~ $\quad \mathbf{G_{3-4}}$
33 $pk \leftarrow \mathrm{gen}_{\mathsf{K}}(sk)$
34 $st_A \leftarrow (pk, k.c^*, k.m, t)$
35 $\mathrm{AC}_A[s_A] \leftarrow (ad, C)$
36 If $s_A = oos_B \wedge (ad, C) = acos_B$: $\quad \mathbf{G_{\geq 3}}$
37 Abort $\quad \mathbf{G_{\geq 3}}$
38 $\mathrm{K}_A[s_A] \leftarrow k.o$
39 $s_A \leftarrow s_A + 1$
40 $i \leftarrow (s_A, \mathtt{S})$ $\quad \mathbf{G_{\geq 2}}$
41 $KT_\star[i] \leftarrow (k.c^*, k.m, t)$ $\quad \mathbf{G_{\geq 2}}$
42 $SK_\star[i] \leftarrow sk$ $\quad \mathbf{G_{\geq 2}}$
43 Return C

Oracle ExposeA
As in KIND

Oracle Reveal(u, j)
As in KIND

Oracle RcvB(ad, C)

44 Require $st_B \neq \bot$
45 If $is_B \wedge \mathrm{AC}_A[r_B] \neq (ad, C)$:
46 $SK_\star[r_B, \mathtt{R}] \leftarrow SK_\star[r_B, \mathtt{S}]$ $\quad \mathbf{G_{\geq 2}}$
47 $KT_\star[r_B, \mathtt{R}] \leftarrow KT_\star[r_B, \mathtt{S}]$ $\quad \mathbf{G_{\geq 2}}$
48 $oos_B \leftarrow r_B$; $acos_B \leftarrow (ad, C)$ $\quad \mathbf{G_{\geq 3}}$
49 $is_B \leftarrow \mathtt{F}$
50 If $r_B \in \mathrm{XP}_A$:
51 $\mathrm{KN}_B \xleftarrow{\cup} [r_B, \ldots]$
52 Else if $(r_B, \mathtt{S}) \notin XSK$: $\quad \mathbf{G_{\geq 5}}$
53 $forge \leftarrow \mathtt{T}$ $\quad \mathbf{G_{\geq 5}}$
54 If not is_B: $\quad \mathbf{G_{\geq 2}}$
55 $i \leftarrow (r_B, \mathtt{R})$ $\quad \mathbf{G_{\geq 2}}$
56 ~~$(sk, k.c, k.m, t) \leftarrow st_B$~~ $\quad \mathbf{G_{<2}}$
57 $(k.c, k.m, t) \leftarrow KT_\star[i]$ $\quad \mathbf{G_{\geq 2}}$
58 ~~$sk \leftarrow SK_\star[i]$~~ $\quad \mathbf{G_{2-4}}$
59 $c \,\|\, ck \,\|\, \tau \leftarrow C$
60 Require $\mathrm{vfy}_{\mathsf{M}}(k.m, ad \,\|\, c \,\|\, ck, \tau)$
61 If $forge$: Abort $\quad \mathbf{G_{\geq 5.2}}$
62 ~~$k \leftarrow \mathrm{dec}(sk, c)$~~ $\quad \mathbf{G_{<3}}$
63 ~~Require $k \neq \bot$~~ $\quad \mathbf{G_{<3}}$
64 Require $\mathrm{dec}(SK_\star[i], c) \neq \bot$ $\quad \mathbf{G_{\geq 3}}$
65 $t \xleftarrow{\cup} ad \,\|\, C$
66 ~~$k.o \,\|\, k.c^* \,\|\, k.m \,\|\, sk \leftarrow \mathrm{H}(k.c, k, t)$~~ $\quad \mathbf{G_{<3}}$
67 $k.o \,\|\, k.c^* \,\|\, k.m \,\|\, sk \leftarrow \mathrm{G}(k.c, t, i, \mathtt{F})$ $\quad \mathbf{G_{\geq 3}}$
68 ~~$st_B \leftarrow (sk, k.c^*, k.m, t)$~~ $\quad \mathbf{G_{<2}}$
69 If is_B: $k.o \leftarrow \diamond$
70 $\mathrm{K}_B[r_B] \leftarrow k.o$
71 $r_B \leftarrow r_B + 1$
72 If not is_B: $\quad \mathbf{G_{\geq 2}}$
73 $i \leftarrow (r_B, \mathtt{R})$ $\quad \mathbf{G_{\geq 2}}$
74 $KT_\star[i] \leftarrow (k.c^*, k.m, t)$ $\quad \mathbf{G_{\geq 2}}$
75 $SK_\star[i] \leftarrow sk$ $\quad \mathbf{G_{\geq 2}}$
76 Return

Oracle ExposeB

77 $\mathrm{KN}_B \xleftarrow{\cup} [r_B, \ldots]$
78 If is_B:
79 $\mathrm{KN}_A \xleftarrow{\cup} [r_B, \ldots]$
80 $\mathtt{U} \leftarrow \mathtt{S}$ $\quad \mathbf{G_{\geq 2}}$
81 $XSK \xleftarrow{\cup} [r_B, \ldots] \times [\mathtt{S}]$ $\quad \mathbf{G_{\geq 2}}$
82 Else: $\mathtt{U} \leftarrow \mathtt{R}$ $\quad \mathbf{G_{\geq 2}}$
83 $(k.c, k.m, t) \leftarrow KT_\star[r_B, \mathtt{U}]$ $\quad \mathbf{G_{\geq 2}}$
84 $sk \leftarrow SK_\star[r_B, \mathtt{U}]$ $\quad \mathbf{G_{\geq 2}}$
85 $st_B \leftarrow (sk, k.c, k.m, t)$ $\quad \mathbf{G_{\geq 2}}$
86 Return st_B

Oracle Challenge(u, j)
As in KIND

Figure 3.15: Proof of URKE.

Game 2 – Synchronous simulation of B

From correctness, we can conclude that B will compute the same values as A after processing the ciphertext from A under the condition that the ciphertext was not manipulated in transmission (A further generates the public key from the secret key and removes the secret key afterwards). Therefore we do not simulate the receiving of B if synchronicity was not disrupted. In order to simulate the exposure of B correctly, we introduce two arrays $SK_\star, KT_\star$ that track A's internal outputs (secret key and symmetric keys with transcript) after sending.

As soon as B receives a manipulated ciphertext, his state is established from these arrays and then the simulation is further computed according to the construction. Indexes of the arrays $SK_\star, KT_\star$ are tuples of a counter and a role indicator. This indicator is set to $[\mathtt{R}]$ for stored values of only B and for the last common values among A and B. We denote incrementing of index $i+1 = (s, \mathtt{U})+1 = (s+1, \mathtt{U})$ such that the counter s is incremented. To shorten and clarify the description, we merge the index parameters in variable i. Thereby i differs from the game internal variable j used in oracles Reveal and Challenge.

Additionally we introduce the array CK and the set XSK. XSK is used to track which secret keys of A's public keys are (potentially) exposed. Thereby the indexes of both the secret key that is actually exposed and all subsequent secret keys that are 'derived' from it are unified in XSK. One secret key is derived from another one if the earlier secret key (in combination with the simultaneously exposed chaining key $k.c$) can be used to obtain all information (in particular the encapsulated key k) to request the random oracle on output of the next secret key. Array CK stores the encapsulated KEM keys k of A for ciphertexts c under the public key with secret key i. This resembles the use of array CK in game OW (see Figure 2.4) such that the reduction can directly use this mapping in the reduction.

Random oracle

We construct our random oracle by defining several procedures that program the simulated function and that can request the output on a provided input. Procedures G and SetO are defined only for the simulation's internal requests and oracle H is provided to the adversary.

The public oracle H is initially defined as a function that randomly samples an output value on the first request of the input value and outputs it on all further requests of this input. In addition to the output, consisting of three symmetric keys and the secret key, a flag *sen* is stored for every entry to trace the requests' origin (i.e., whether A's simulation—the sender—initially requested the random oracle for that entry or not).

The description of the random oracle from Figure 3.16 follows the same principle as the one for the proof of SRKE (see Figure 3.19).

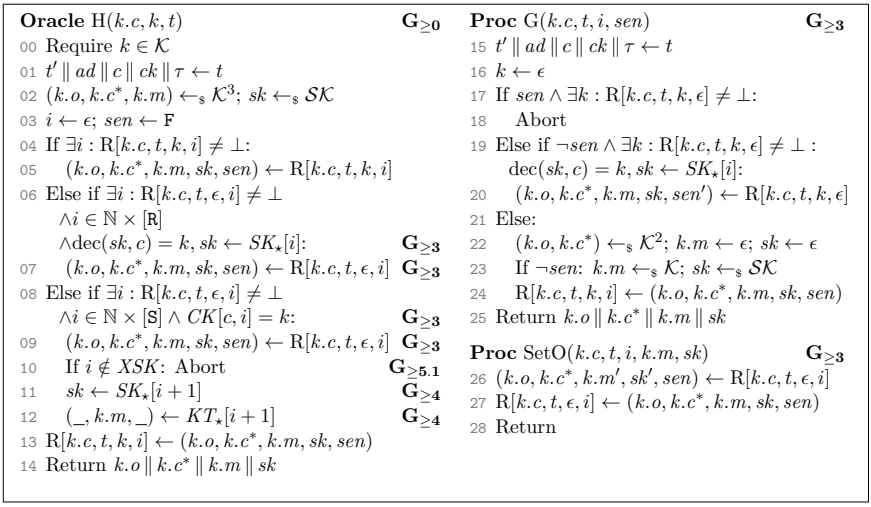

Figure 3.16: Random oracle description for proof of URKE.

Game 3 – Internal access to random oracle

We introduce the internal procedures to request the random oracle. These procedures provide the simulation the opportunity to request the random oracle output on an undefined input key k. The index for the internal 'storage' array of the random oracle is thereby defined by chaining key $k.c$ and transcript t—hence implicitly by the last ciphertext c—as inputs to the random oracle, as well as the index of the key pair for which this last ciphertext is encapsulated. Hence, for internally requested ciphertexts c, the correct but unknown key k is implicitly programmed as input into the random oracle.

By requesting the random oracle without providing the decapsulated key, the decapsulation and hence the previous secret key of B are not needed to simulate receiving anymore. In order to correctly simulate, the output of the decapsulation still needs to be compared to the $\perp$ symbol. When reducing to OW security, this comparison can be conducted by requesting the game's Check oracle without using the respective secret key explicitly (see Figure 2.4).

External queries to the random oracle H are stored in entries of the simulation's array R, indexed by input tuples $(k.c, k, t)$ together with parameter $i = \epsilon$ to signal that the entries are not associated to a secret key that is used by the simulation. The output values are sampled uniformly at random for an initial request and afterwards reaccessed from the described array entry. Initial internal queries are stored with an empty string ϵ at the index parameter position of the input KEM key k. Additionally, the index of the secret key i of the previous KEM operation is set as array index parameter (for A's simulation: index of secret key for public key with which encapsulation outputting the last c in t was executed; for B's simulation: index of secret key with which decapsulation would have been called). For initial internal queries of A's simulation, only the output session key $k.o$ and the output chaining key $k.c^*$ are sampled and set within G. The remaining outputs are sampled independently and set by the procedure SetO. This makes no difference regarding the simulation but will be important for lazy sampling in game G_4. For the simulation of B's

internal queries, all outputs values are sampled at once.

In order to show how correctness and output distribution of the random oracle are preserved under the above described behavior, we first consider how the sequence of 1) an external request before 2) an internal request to the same entry is processed and then vice versa. Note that equality of queries cannot be verified via the input values $(k.c, k, t)$ since k is not specified for internal requests directly.

Two successive *external* requests are still processed as before: if the respective entry does not exist yet, it is generated, otherwise it is accessed and returned.

Two successive *internal* requests to the same entry only occur if the adversary is able to predict a collision key of A. Note that, since the inputs to the random oracle include the transcript, two successive requests with the same transcript cannot be made by only A or only B, respectively. As the simulation of B queries the random oracle only out of sync, a random oracle request by A to an entry that was created by B only occurs if the adversary manages to keep B's transcript equal to A's transcript when A and B are out of sync. This only happens if A's and B's transcripts equal in sync, then B receives an adversarially crafted ciphertext c that brings him out of sync, and finally A sends the same ciphertext c such that B is out of sync but A's and B's transcripts equal again. In order to achieve this, the adversary must predict collision key ck in A's ciphertext c when priorly crafting and sending the (same) ciphertext to B. Hence, aborting in line 37 (Figure 3.15) can be bounded by $1/|\mathcal{K}|$. After this abort-event, the same entry cannot be requested internally first by one of A or B and then by the other one (as thereby B's transcript definitely differs from A's transcript out of sync).

An internal random oracle request of A's simulation is marked by the last parameter of G set true (i.e., $sen = \mathtt{T}$). For requests of this form, the game aborts if there already exists any random oracle entry with the same transcript and chaining key (independent of the KEM key input k). Since the last ciphertext c in transcript t contains a random collision key, sampled right before the internal random oracle request—and thereby not known to the adversary—, the probability

of the abortion in line 18 can be bounded by $q_{\mathrm{H}}/|\mathcal{K}|$. (Note that due to the input transcript each external random oracle query can only collide with exactly one internal random oracle query.)

For internal requests of B's simulation, previous random oracle entries with the same transcript and chaining key are validated with respect to the tuple (c, i, k) where c is the last ciphertext in t, k is the input KEM key of an existing random oracle entry queried externally, and i is the index of the secret key with which G is invoked. If k can be decapsulated from c with $SK_\star[i]$, then the existing random oracle entry equals the one that was intended to be queried by the simulation of B. Consequently all output values are taken from the found external entry and returned by G.

If there exists an entry made internally (and hence without specified k) for an external random oracle request, a validation with respect to the tuple (c, i, k) is conducted as well. Thereby external request and internal entry are equal if the ciphertext of both inputs can be decapsulated by secret key with index i from the internal entry's index parameters to the input key k from the external request. This validation is split into entries made by B (Figure 3.16 line 06) and entries made by the A with secret key i (tracked by array CK; Figure 3.16 line 08).

Array CK represents oracle Solve from the OW game (see Figure 2.4) that can only be queried for unexposed secret keys. In order to access all ciphertext-secret key pairs of A, the reduction will manually fill array CK for exposed secret keys and the Solve oracle is requested for (the remaining) unexposed secret keys. Thereby the simulation of array CK is sound.

Due to the abortion in the random oracle in line 18, a random oracle entry that is internally requested by A was not requested by the adversary before. Similarly, due to the abortion during A's sending in line 37, all random oracle queries by B (out of sync) differ from A's random oracle queries. As argued before, the probability of an abortion in game G_3 can be upper bounded with q_{H}, being the number of external random oracle queries, and $\mathcal{K}$, being the key space of the

collision key:

$$\mathrm{Adv}^{G_2,G_3}(\mathcal{D}) \leq \frac{q_{\mathrm{H}}+1}{|\mathcal{K}|}$$

Game 4 – Random oracle with lazy sampling

In game G_4 we stop to set secret key and MAC key as part of the output values of the internal random oracle requests of A's sending. Instead the actual output of the external random oracle is instantly set as soon as the adversary requests it.

Random oracle entries, defined by the internal requests of B, are still equipped with all output values as before. As we will show in the next game, these outputs will not be challengeable anymore. This game hop includes only cosmetic changes and is, hence, undetectable by adversaries.

Game 5 – Abortion In game G_5 we finally abort if the adversary queries the random oracle for entries that reveal a challengeable key.

At first, game $G_{5.1}$ is aborted if the adversary externally queries an entry that was made by A with a ciphertext (encoded in t) that is designated for an unexposed secret key (see Figure 3.16 line 10). Since only entries made by A result in this abortion, it only occurs if the adversary was able to derive a key k from a ciphertext c sent by A to correctly request the random oracle.

To reduce the probability of this abortion event to OW security of KEM K, the secret key with which the key k can be decapsulated from the ciphertext c must not have been used by the simulation. This holds because on the one hand, the secret key was not exposed (by the abortion condition) and on the other hand, the random oracle entry that would have contained this secret key as output was not requested before, because otherwise the simulation would have been aborted before.

It is now important to observe that the exposed keys in XSK correspond to the adversarially known established keys in KN_A (i.e., $\mathrm{KN}_A = XSK$). Hence for the game to be aborted, the random or-

acle must be requested for any challengeable established key of A. Consequently either the adversary did not request the random oracle for an entry with which it would know an established key of A or the game aborted and thereby game OW is won.

$$\mathrm{Adv}^{G_5, G_{5.1}}(\mathcal{D}) \leq \mathrm{Adv}^{\mathrm{ow}}_{\mathsf{K}}(\mathcal{B}_{\mathsf{K}})$$

After game $G_{5.1}$, for gaining an advantage in winning the game, the adversary can only gather information via requesting the random oracle for asynchronously established keys of B.

Therefore consider that at receiving the first manipulated ciphertext, the following conditions need to hold for $\mathit{forge} = \mathtt{T}$ (see lines 50–53): 1) A was not exposed to be impersonated towards B and 2) B's current secret key was not marked to be exposed. In addition, these conditions imply that 3) the last *common* random oracle query of A and B was not requested by the adversary. Condition 3 holds because condition 2 implies that A is challengeable and an external query to a random oracle entry outputting a challengeable key would have resulted in an abortion of game $G_{5.1}$.

As a result of these conditions, the simulation did not leak the MAC key in B's state at receiving the manipulated ciphertext—except that it was used to generate the MAC tag of a potentially sent ciphertext of A—and the MAC key was sampled uniformly at random. Now distinguishing between games $G_{5.1}$ and $G_{5.2}$ can be reduced to SUF security of MAC M because $G_{5.2}$ only aborts if the MAC tag was valid but the ciphertext was manipulated and, as shown above, the MAC key was not provided to the adversary. Hence in order to distinguish between these two games, a distinguisher needs to provide a forged MAC tag. Therefore a distinguisher's advantage can be bounded by:

$$\mathrm{Adv}^{G_{5.1}, G_{5.2}}(\mathcal{D}) \leq \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}})$$

If $G_{5.2}$ did not abort, B cannot be challenged because either one of conditions 1 or 2 were violated and therefore the remaining keys of B are set to be known (see lines 51 and 79), or the state was erased due to an invalid MAC tag.

Finally the adversary can win game $G_{5.2}$ with probability 1/2 because no information on a challengeable key of either A or B can be gained by the adversary.

$$\mathrm{Adv}^{G_{5.2}}(\mathcal{A}) = 0$$

Proof result Taking the bounds drawn in the game hops above provides us the upper bound of the advantage that an adversary has in the URKE KIND game depending on the advantage of the adversaries $\mathcal{B}_{\mathsf{K}}$ and $\mathcal{B}_{\mathsf{M}}$:

$$\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{R}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{ow}}_{\mathsf{K}}(\mathcal{B}_{\mathsf{K}}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}}) + \frac{q_{\mathrm{H}} + 1}{|\mathcal{K}|}$$

Please note that the URKE proof makes no use of the Up_R oracle of the KUOW game and therefore a generic CCA secure KEM suffices to reach URKE (with a loss factor of q_{SndA}). However, the update algorithm could be used instead of generating new key pairs after each sending and receiving since it provides the same functionality and security but A would never learn the secret key to her public key. This property plays a crucial role in case A's random coins are attacked during sending. We will consider this attack vector in depth, and use the above sketched solution, in Chapter 4.

3.11 Proof of SRKE

Model and construction of SRKE are very similar to the model and construction of URKE, respectively. Therefore, the proof follows the same idea. The exact game hops are, however, more complex because of the more sophisticated employed primitives in the SRKE construction and the management of variables due to the extended communication setup (both parties can send and receive anytime).

Even though the security of URKE is implied by the security of SRKE, for didactic reasons we provide both proofs.

Overview

In addition to the proof of URKE, SRKE employs an abortion rule in game G_1 in case of a signature forgery. All remaining game hops follow the same idea as the URKE proof: G_2 restricts B's simulation to sending and *out of sync* receiving. Receiving *in sync* of B is simulated by the computations of A's sending—which by correctness of the construction has the same result. G_3 programs the random oracle with internal procedures for the simulation without the knowledge of encapsulated kuKEM keys such that sent ciphertexts can include embedded kuKEM challenges. To reduce the adversary's random oracle queries to the solution of the employed hardness assumptions, G_4 starts to lazy sample the random oracle outputs—including the next secret key and MAC key. Finally, G_5 aborts on bad events which are proven to only occur if the adversary broke one of the hardness assumptions.

Figure 3.17 and Figure 3.18 depict the SRKE $\mathrm{KIND}_{\mathsf{R}}$ game (see Figure 3.9) including our SRKE construction from Figure 3.10 and the game hops described below. Figure 3.17 includes the initialization of game and scheme, and the communication from B to A with a helper procedure GetSK. The opposite communication direction is depicted in Figure 3.18 with the oracles for the adversary to expose, reveal, and challenge. The description of the random oracle can be found in Figure 3.19.

Game 1 – Excluding signature forgeries

Game G_1 aborts if the adversary successfully forges a signature to drift the parties' states out of sync. Thereby distinguishing between the original game $\mathrm{KIND}_{\mathsf{R}}$ and G_1 can be reduced to the SUF security of the one-time signature scheme. In order to do so, the reduction replaces key-pair generation, signing, and verification algorithms by the SUF game's oracles. To correctly simulate exposures of B that also reveal B's signing keys, the reduction can use the expose oracle from our multi-instance SUF notion. The signer key *sgk* for which

Simulation $\mathcal{S}^k_{\mathsf{R}}(\mathcal{A})$

00 $\mathrm{R}[\cdot] \leftarrow \bot$ $\quad \mathbf{G}_{\geq \mathbf{0}}$
01 $s_A \leftarrow 0$; $r_B \leftarrow 0$; $is_A \leftarrow \mathtt{T}$
02 $s_B \leftarrow 0$; $r_A \leftarrow 0$; $is_B \leftarrow \mathtt{T}$
03 $e_A \leftarrow 0$; $\mathrm{EP}_A[\cdot] \leftarrow \bot$
04 $\mathrm{E}_B^{\lhd} \leftarrow 0$; $\mathrm{E}_B^{\rhd} \leftarrow 0$
05 $\mathrm{AC}_A[\cdot] \leftarrow \bot$; $\mathrm{K}_B[\cdot] \leftarrow \bot$
06 $\mathrm{AC}_B[\cdot] \leftarrow \bot$; $\mathrm{K}_A[\cdot] \leftarrow \bot$
07 $\mathrm{XP}_A \leftarrow \emptyset$; $\mathrm{KN}_A \leftarrow \emptyset$; $\mathrm{CH}_A \leftarrow \emptyset$
08 $\mathrm{XP}_B \leftarrow \emptyset$; $\mathrm{KN}_B \leftarrow \emptyset$; $\mathrm{CH}_B \leftarrow \emptyset$
09 $oos_B \leftarrow \infty$; $forge_B \leftarrow \mathtt{F}$; $acos_B \leftarrow \bot$ $\quad \mathbf{G}_{\geq \mathbf{1}}$
10 $SK_\star[\cdot] \leftarrow \bot$; $KT_\star[\cdot] \leftarrow \bot$; $P[\cdot] \leftarrow \bot$ $\quad \mathbf{G}_{\geq \mathbf{2}}$
11 $es_A \leftarrow 0$; $er_B \leftarrow 0$; $ims \leftarrow \mathtt{F}$ $\quad \mathbf{G}_{\geq \mathbf{2}}$
12 $XSK \leftarrow \emptyset$; $\Gamma[\cdot] \leftarrow 0$; $CK[\cdot] \leftarrow \bot$ $\quad \mathbf{G}_{\geq \mathbf{2}}$
13 $oos_A \leftarrow \infty$; $forge_A \leftarrow \mathtt{F}$; $acos_A \leftarrow \bot$ $\quad \mathbf{G}_{\geq \mathbf{3}}$
14 $(sgk_\star, vfk) \leftarrow_{\$} \mathrm{gen}_{\mathsf{S}}$ $\quad \mathbf{G}_{\geq \mathbf{2}}\star$
15 $(sk, pk) \leftarrow_{\$} \mathrm{gen}_{\mathsf{K}}$
16 $(k.c, k.m) \leftarrow_{\$} \mathcal{K}^2$; $t \leftarrow \epsilon$
17 $E^{\lhd} \leftarrow 0$; $E^{\rhd} \leftarrow 0$
18 $s \leftarrow 0$; ~~$r \leftarrow 0$~~ $\quad \mathbf{G}_{<\mathbf{2}} \not\equiv$
19 $E_\star^{\lhd} \leftarrow 0$; $E_\star^{\rhd} \leftarrow 0$ $\quad \mathbf{G}_{\geq \mathbf{2}}\star$
20 $PK[\cdot] \leftarrow \bot$; $PK[0] \leftarrow pk$
21 ~~$SK[\cdot] \leftarrow \bot$; $SK[0] \leftarrow sk$~~ $\quad \mathbf{G}_{<\mathbf{2}}$
22 $L_A[\cdot] \leftarrow \bot$; $L_A[0] \leftarrow \diamond$
23 $L_{B\star}[\cdot] \leftarrow \bot$ $\quad \mathbf{G}_{\geq \mathbf{2}}\star$
24 $st_A \leftarrow (PK, E, s, L_A, vfk, k.c, k.m, t)$
25 ~~$st_B \leftarrow (SK, E, r, L_B, sgk, k.c, k.m, t)$~~ $\quad \mathbf{G}_{<\mathbf{2}}$
26 $KT_\star[0,0,0,\mathtt{S}] \leftarrow (k.c, k.m, \epsilon)$ $\quad \mathbf{G}_{\geq \mathbf{2}}$
27 $SK_\star[0,0,0,\mathtt{S}] \leftarrow (sk, 0)$ $\quad \mathbf{G}_{\geq \mathbf{2}}$
28 $b' \leftarrow_{\$} \mathcal{A}$
29 Require $\mathrm{KN}_A \cap \mathrm{CH}_A = \emptyset$
30 Require $\mathrm{KN}_B \cap \mathrm{CH}_B = \emptyset$
31 Stop with b'

Oracle SndB(ad)

32 Require $st_B \neq \bot$
33 ~~$(SK, E, r, L, sgk, k.c, k.m, t) \leftarrow st_B$~~ $\quad \mathbf{G}_{<\mathbf{2}}$
34 $(sk^*, pk^*) \leftarrow_{\$} \mathrm{gen}_{\mathsf{K}}$
35 $(sgk^*, vfk^*) \leftarrow_{\$} \mathrm{gen}_{\mathsf{S}}$
36 $E_\star^{\rhd} \leftarrow E_\star^{\rhd} + 1$; ~~$SK[E^{\rhd}] \leftarrow sk^*$~~ $\quad \mathbf{G}_{<\mathbf{2}}\star$
37 $ck \leftarrow_{\$} \mathcal{K}$; $C \leftarrow r_B \,\|\, pk^* \,\|\, vfk^* \,\|\, ck$ $\quad \mathbf{G}_{\geq \mathbf{2}}\star$
38 $\sigma \leftarrow_{\$} \mathrm{sgn}(sgk_\star, ad \,\|\, C)$ $\quad \mathbf{G}_{\geq \mathbf{2}}\star$
39 $sgk_\star \leftarrow sgk^*$ $\quad \mathbf{G}_{\geq \mathbf{2}}\star$
40 $C \leftarrow C \,\|\, \sigma$; $L_{B\star}[E_\star^{\rhd}] \leftarrow \triangleleft \,\|\, ad \,\|\, C$ $\quad \mathbf{G}_{\geq \mathbf{2}}\star$
41 ~~$st_B \leftarrow (SK, E, r, L, sgk, k.c, k.m, t)$~~ $\quad \mathbf{G}_{<\mathbf{2}}$
42 If $s_B = oos_A \wedge (ad, C) = acos_A$: $\quad \mathbf{G}_{\geq \mathbf{3}}$
43 Abort $\quad \mathbf{G}_{\geq \mathbf{3}}$
44 If is_B:
45 $\mathrm{AC}_B[s_B] \leftarrow (ad, C)$
46 $\mathrm{E}_B^{\rhd} \leftarrow \mathrm{E}_B^{\rhd} + 1$
47 $s_B \leftarrow s_B + 1$
48 If is_B: $SK_\star[E_\star^{\rhd}, 0, 0, \mathtt{S}] \leftarrow (sk^*, s_B)$ $\quad \mathbf{G}_{\geq \mathbf{2}}$
49 Else: $SK_\star[E_\star^{\rhd}, 0, 0, \mathtt{R}] \leftarrow (sk^*, s_B)$ $\quad \mathbf{G}_{\geq \mathbf{2}}$
50 Return C

Oracle RcvA(ad, c)

51 Require $st_A \neq \bot$
52 If $is_A \wedge \mathrm{AC}_B[r_A] \neq (ad, c)$:
53 $is_A \leftarrow \mathtt{F}$
54 $oos_A \leftarrow r_A$; $acos_A \leftarrow (ad, c)$ $\quad \mathbf{G}_{\geq \mathbf{3}}$
55 If $r_A \in \mathrm{XP}_B$:
56 $\mathrm{KN}_A \xleftarrow{\cup} \mathbb{N} \times [\, s_A, \ldots]$
57 $ims \leftarrow \mathtt{T}$ $\quad \mathbf{G}_{\geq \mathbf{2}}$
58 Else if $oos_B > r_A$: $forge_B \leftarrow \mathtt{T}$ $\quad \mathbf{G}_{\geq \mathbf{1}}$
59 If is_A: $e_A \leftarrow e_A + 1$
60 $(PK, E, s, L, vfk, k.c, k.m, t) \leftarrow st_A$
61 $t \xleftarrow{} \triangleleft \,\|\, ad \,\|\, C$; $C \,\|\, \sigma \leftarrow C$
62 Require $\mathrm{vfy}_{\mathsf{S}}(vfk, ad \,\|\, C, \sigma)$
63 If $forge_B$: Abort $\quad \mathbf{G}_{\geq \mathbf{1}}$
64 $r \,\|\, pk^* \,\|\, vfk \,\|\, ck \leftarrow C$
65 Require $L[r] \neq \bot$
66 $L[\ldots, (r-1)] \leftarrow \bot$; $L[r] \leftarrow \diamond$
67 For $s' \leftarrow r + 1$ to s:
68 $pk^* \leftarrow \mathrm{up}(pk^*, L[s'])$
69 $\Gamma[E^{\rhd} + 1, \mathtt{S}] \leftarrow \Gamma[E^{\rhd} + 1, \mathtt{S}] + 1$ $\quad \mathbf{G}_{\geq \mathbf{2}}$
70 $E^{\rhd} \leftarrow E^{\rhd} + 1$; $PK[E^{\rhd}] \leftarrow pk^*$
71 $st_A \leftarrow (PK, E, s, L, vfk, k.c, k.m, t)$
72 If $st_A = \bot$: Return $\bot$
73 $r_A \leftarrow r_A + 1$
74 Return

Proc GetSK($\mathtt{U}_1, \mathtt{U}_2$) $\quad \mathbf{G}_{\geq \mathbf{2}}$

75 If $\mathtt{U}_1 = \mathtt{S}$: $(\_\_, L_A, \_\_) \leftarrow st_A$; $P \leftarrow L_A$
76 $SK[\cdot] \leftarrow \bot$
77 $(sk, s) \leftarrow SK_\star[E_\star^{\lhd}, er_B, 0, \mathtt{U}_1]$
78 $SK[E_\star^{\lhd}] \leftarrow sk$; $SK_\star[E_\star^{\lhd}, er_B, 0, \mathtt{U}_2] \leftarrow (sk, s)$
79 For ε from $E_\star^{\lhd} + 1$ to $E_\star^{\rhd}$:
80 $l \leftarrow \max(\ell : SK_\star[\varepsilon, 0, \ell, \mathtt{U}_1] \neq \bot)$
81 $(sk, s) \leftarrow SK_\star[\varepsilon, 0, l, \mathtt{U}_1]$
82 $SK_\star[\varepsilon, 0, l, \mathtt{U}_2] \leftarrow (sk, s)$
83 For $\ell \leftarrow l + 1$ to $r_B - s$:
84 $p \leftarrow P[r_B - s + \ell]$
85 $sk \leftarrow_{\$} \mathrm{up}(sk, p)$
86 $SK_\star[\varepsilon, 0, \ell, \mathtt{U}_2] \leftarrow (sk, s)$
87 $SK[\varepsilon] \leftarrow sk$
88 $\Gamma[\varepsilon, \mathtt{R}] \leftarrow r_B - s$
89 Return SK

Note on interval variables $\mathrm{E}_B, E, E_\star$:
E_B is the KIND game's interval for B which stops increasing *out of sync*,
E denotes the intervals in A's and B's states *locally*, and
$E_\star$ is the global substitution for the interval in B's state (for $G_{\geq 2}$).

Figure 3.17: Proof of SRKE containing initialization, communication from B to A, and two helper procedures.

Oracle SndA(ad)
00 Require $st_A \neq \bot$
01 $(PK, E, s, L, vfk, k.c, k.m, t) \leftarrow st_A$
02 $i \leftarrow (E^{\lhd}, es_A, 0, \mathtt{S})$ $\mathbf{G}_{\geq 2}$
03 If $E^{\lhd} \neq E^{\rhd}$: $es_A \leftarrow 0$ $\mathbf{G}_{\geq 2}$
04 $k^* \leftarrow \epsilon$; $ck \leftarrow_{\$} \mathcal{K}$; $C \leftarrow E^{\rhd} \,\|\, ck$
05 For $e' \leftarrow E^{\lhd}$ to $E^{\rhd}$:
06 $(k, c) \leftarrow_{\$} \mathrm{enc}(PK[e'])$
07 $CK[c, i] \leftarrow k$ $\mathbf{G}_{\geq 2}$
08 ~~$k^* \xleftarrow{\cup} k$~~; $C \xleftarrow{\cup} c$ $\mathbf{G}_{<3}$ ~~$k^* \xleftarrow{\cup} k$~~
09 If *ims*: $k^* \xleftarrow{\cup} k$ $\mathbf{G}_{\geq 3}$
10 $I \xleftarrow{\cup} i$; $i \leftarrow (e' + 1, 0, \Gamma[e' + 1, \mathtt{S}], \mathtt{S})$ $\mathbf{G}_{\geq 2}$
11 $\tau \leftarrow \mathrm{tag}(k.m, ad \,\|\, C)$
12 $C \xleftarrow{\cup} \tau$; $t \xleftarrow{\cup} \triangleright \,\|\, ad \,\|\, C$
13 If *ims*: $y \,\|\, k.m \,\|\, sk \leftarrow \mathrm{H}(k.c, k^*, t)$ $\mathbf{G}_{\geq 3}$ If *ims*
14 Else: $\mathbf{G}_{\geq 3}$
15 $y \,\|\, \_\_ \leftarrow \mathrm{G}(k.c, t, I, \mathtt{T})$ $\mathbf{G}_{\geq 3}$
16 $k.m \leftarrow_{\$} \mathcal{K}$; $sk \leftarrow_{\$} \mathcal{SK}$ $\mathbf{G}_{\geq 3}$
17 ~~SetO$(k.c, t, I, k.m, sk)$~~ $\mathbf{G}_{3-4}$
18 $k.o \,\|\, k.c \leftarrow y$
19 $pk \leftarrow \mathrm{gen}_{\mathsf{K}}(sk)$
20 $PK[\ldots, (E^{\rhd} - 1)] \leftarrow \bot$; $PK[E^{\rhd}] \leftarrow pk$
21 $E^{\lhd} \leftarrow E^{\rhd}$; $s \leftarrow s + 1$; $L[s] \leftarrow ad \,\|\, C$
22 $st_A \leftarrow (PK, E, s, L, vfk, k.c, k.m, t)$
23 If $s_A = oos_B \wedge (ad, C) = acos_B$: $\mathbf{G}_{\geq 3}$
24 Abort $\mathbf{G}_{\geq 3}$
25 If is_A:
26 $\mathrm{AC}_A[s_A] \leftarrow (ad, C)$
27 $\mathrm{EP}_A[s_A] \leftarrow e_A$
28 $\mathrm{K}_A[e_A, s_A] \leftarrow k.o$
29 $s_A \leftarrow s_A + 1$
30 $es_A \leftarrow es_A + 1$; $i \leftarrow (E^{\rhd}, es_A, 0, \mathtt{S})$ $\mathbf{G}_{\geq 2}$
31 If not *ims*: $\mathbf{G}_{\geq 2}$
32 $KT_\star[i] \leftarrow (k.c, k.m, t)$ $\mathbf{G}_{\geq 2}$
33 $SK_\star[i] \leftarrow (sk, \bot)$ $\mathbf{G}_{\geq 2}$
34 Return C

Oracle ExposeA
35 If is_A: $\mathrm{XP}_A \xleftarrow{\cup} \{s_A\}$
36 Return st_A

Oracle ExposeB
37 $\mathrm{KN}_B \xleftarrow{\cup} [\mathrm{E}_B^{\lhd} \,..\, \mathrm{E}_B^{\rhd}] \times [\, r_B, \ldots\,]$
38 If is_B:
39 $\mathrm{XP}_B \xleftarrow{\cup} \{s_B\}$
40 $\mathrm{KN}_A \xleftarrow{\cup} [\mathrm{E}_B^{\lhd} \,..\, \mathrm{E}_B^{\rhd}] \times [\, r_B, \ldots\,]$
41 $\mathtt{U} \leftarrow \mathtt{S}$ $\mathbf{G}_{\geq 2}$
42 $XSK \xleftarrow{\cup} [E_\star^{\lhd}] \times [\, er_B, \ldots\,] \times \mathbb{N} \times [\mathtt{S}]$ $\mathbf{G}_{\geq 2}$
43 $XSK \xleftarrow{\cup} [E_\star^{\lhd} + 1 \,..\, E_\star^{\rhd}] \times \mathbb{N}^2 \times [\mathtt{S}]$ $\mathbf{G}_{\geq 2}$
44 Else: $\mathtt{U} \leftarrow \mathtt{R}$ $\mathbf{G}_{\geq 2}$
45 $(k.c, k.m, t) \leftarrow KT_\star[E_\star^{\lhd}, er_B, 0, \mathtt{U}]$ $\mathbf{G}_{\geq 2}$
46 $SK \leftarrow \mathrm{GetSK}(\mathtt{U}, \mathtt{U})$ $\mathbf{G}_{\geq 2}$
47 $st_B \leftarrow (SK, E_\star, r_B, L_{B\star}, sgk_\star, k.c, k.m, t)$ $\mathbf{G}_{\geq 2}$
48 Return st_B

Oracle Reveal(u, j)
as in URKE (Fig. 3.6)

Oracle Challenge(u, j)
as in URKE (Fig. 3.6)

Oracle RcvB(ad, C)
49 Require $st_B \neq \bot$
50 If $is_B \wedge \mathrm{AC}_A[r_B] \neq (ad, C)$:
51 $is_B \leftarrow \mathtt{F}$
52 $oos_B \leftarrow s_B$; $acos_B \leftarrow (ad, C)$ $\mathbf{G}_{\geq 1}$
53 $\mathrm{GetSK}(\mathtt{S}, \mathtt{R})$ $\mathbf{G}_{\geq 2}$
54 $KT_\star[E_\star^{\lhd}, er_B, 0, \mathtt{R}] \leftarrow KT_\star[E_\star^{\lhd}, er_B, 0, \mathtt{S}]$ $\mathbf{G}_{\geq 2}$
55 If $r_B \in \mathrm{XP}_A$:
56 $\mathrm{KN}_B \xleftarrow{\cup} \mathbb{N} \times [\, r_B, \ldots\,]$
57 Else if $(E_\star^{\lhd}, er_B, 0, \mathtt{S}) \notin XSK$: $\mathbf{G}_{\geq 5}$
58 $forge_A \leftarrow \mathtt{T}$ $\mathbf{G}_{\geq 5}$
59 If $is_B \wedge \mathrm{E}_B^{\lhd} \neq \mathrm{EP}_A[r_B]$:
60 $\mathrm{E}_B^{\lhd} \leftarrow \mathrm{EP}_A[r_B]$
61 $E_\star^{\lhd} \leftarrow \mathrm{EP}_A[r_B]$; $er_B \leftarrow 0$ $\mathbf{G}_{\geq 2}$
62 If not is_B: $\mathbf{G}_{\geq 2}$
63 $i \leftarrow (E_\star^{\lhd}, er_B, 0, \mathtt{R})$ $\mathbf{G}_{\geq 2}$
64 ~~$(SK, E, r, L, sgk, k.c, k.m, t) \leftarrow st_B$~~ $\mathbf{G}_{<2}$
65 $(k.c, k.m, t) \leftarrow KT_\star[i]$ $\mathbf{G}_{\geq 2}$
66 $SK \leftarrow \mathrm{GetSK}(\mathtt{R}, \mathtt{R})$ $\mathbf{G}_{\geq 2}$
67 $t^* \leftarrow ad \,\|\, C$; $C \,\|\, \tau \leftarrow C$
68 Require $\mathrm{vfy}_{\mathsf{M}}(k.m, ad \,\|\, C, \tau)$
69 If $forge_A$: Abort $\mathbf{G}_{\geq 5.4}$
70 $k^* \leftarrow \epsilon$; $e \,\|\, ck \,\|\, C \leftarrow C$
71 Require $E_\star^{\lhd} \leq e \leq E_\star^{\rhd}$ $\mathbf{G}_{\geq 2}$ ⋆
72 If $E_\star^{\lhd} \neq e$: $er_B \leftarrow 0$ $\mathbf{G}_{\geq 2}$
73 $t \xleftarrow{\cup} L_{B\star}[E_\star^{\lhd} + 1] \,\|\, \ldots \,\|\, L[e]$ $\mathbf{G}_{\geq 2}$ ⋆
74 $L_{B\star}[\ldots, e] \leftarrow \bot$ $\mathbf{G}_{\geq 2}$ ⋆
75 For $e' \leftarrow E_\star^{\lhd}$ to e: $\mathbf{G}_{\geq 2}$ ⋆
76 $c \,\|\, C \leftarrow C$
77 ~~$k \leftarrow \mathrm{dec}(SK[e'], c)$~~ $\mathbf{G}_{<3}$
78 ~~Require $k \neq \bot$~~ $\mathbf{G}_{<3}$
79 Require $\mathrm{dec}(SK_\star[i], c) \neq \bot$ $\mathbf{G}_{\geq 3}$
80 ~~$k^* \xleftarrow{\cup} k$~~ $\mathbf{G}_{<3}$
81 $I \xleftarrow{\cup} i$; $i \leftarrow (e' + 1, 0, \Gamma[e' + 1, \mathtt{R}], \mathtt{R})$ $\mathbf{G}_{\geq 2}$
82 $t \xleftarrow{\cup} \triangleright \,\|\, t^*$
83 ~~$k.o \,\|\, k.c \,\|\, k.m \,\|\, sk \leftarrow \mathrm{H}(k.c, k^*, t)$~~ $\mathbf{G}_{<3}$
84 $k.o \,\|\, k.c \,\|\, k.m \,\|\, sk \leftarrow \mathrm{G}(k.c, t, I, \mathtt{F})$ $\mathbf{G}_{\geq 3}$
85 ~~$SK[\ldots, (e - 1)] \leftarrow \bot$; $SK[e] \leftarrow sk$~~ $\mathbf{G}_{<2}$
86 ~~For $e' \leftarrow e + 1$ to $E^{\rhd}$:~~ $\mathbf{G}_{<2}$
87 ~~$SK[e'] \leftarrow \mathrm{up}(SK[e'], t^*)$~~ $\mathbf{G}_{<2}$
88 $E^{\lhd} \leftarrow e$; ~~$r \leftarrow r + 1$~~ $\mathbf{G}_{<2}$ ↱
89 ~~$st_B \leftarrow (SK, E, r, L, sgk, k.c, k.m, t)$~~ $\mathbf{G}_{<2}$
90 If $st_B = \bot$: Return $\bot$
91 If is_B: $k.o \leftarrow \diamond$
92 $\mathrm{K}_B[\mathrm{E}_B^{\lhd}, r_B] \leftarrow k.o$
93 $r_B \leftarrow r_B + 1$
94 $er_B \leftarrow er_B + 1$ $\mathbf{G}_{\geq 2}$
95 If not is_B: $\mathbf{G}_{\geq 2}$
96 $i \leftarrow (E_\star^{\lhd}, er_B, 0, \mathtt{R})$ $\mathbf{G}_{\geq 2}$
97 $KT_\star[i] \leftarrow (k.c, k.m, t)$ $\mathbf{G}_{\geq 2}$
98 $SK_\star[i] \leftarrow (sk, \bot)$ $\mathbf{G}_{\geq 2}$
99 $P[r_B] \leftarrow t^*$ $\mathbf{G}_{\geq 2}$
100 $\mathrm{GetSK}(\mathtt{R}, \mathtt{R})$ $\mathbf{G}_{\geq 2}$
101 Return

Figure 3.18: Proof of SRKE considering communication from A to B and the remaining oracles of $\mathrm{KIND}_{\mathsf{R}}$.

the adversary provides a forgery—to distinguish within the first game hop—was not exposed because otherwise either the receive counter r_A is in the set of exposed counters XP_B or A was out of sync earlier such that $oos_B \leq r_A$ holds and therefore $forge_B$ would not be set (see lines 55–58).

The advantage of an adversary $\mathcal{D}$ to distinguish between the original game and G_1 can be upper bounded by:

$$\mathrm{Adv}_{\mathsf{R}}^{\mathrm{kind},G_1}(\mathcal{D}) \leq \mathrm{Adv}_{\mathsf{S}}^{\mathrm{suf}}(\mathcal{B}_{\mathsf{S}})$$

Game 2 – Synchronous simulation of B

Using the correctness of our construction, we can again simulate the receiving of unmodified ciphertexts by B with the simulation of A. Therefore we remove B's state in game G_2 and trace the respective variables with global arrays and counters.

To fully understand this game hop, we divide its description on the basis of the introduced and used variables. At first the counters $(E_\star^{|<}, E_\star^{>|}, es_A, er_B)$—and thereby the indexing scheme of the introduced arrays—are explained. Then the usage of the arrays, tracing secret keys ($SK_\star$) and symmetric keys ($KT_\star$), are presented. The array SK in B's state is not simulated directly with one array but instead compiled from the array of *all* secret keys $SK_\star$ as soon as it is needed. Due to the complexity of the compilation, the simulation of array SK (under usage of arrays $SK_\star, \Gamma$ and procedure GetSK) is explained separately. Conclusively, this game introduces an array XSK and a flag ims to comprehensively indicate which secret keys are exposed and how this influences an impersonation towards the sender.

Defining global variables and counters The interval of epochs E, the array of sent ciphertext-associated-data pairs L_B, and the signing key sgk in B's state are declared as global variables to simulate exposures without maintaining B's state as a whole. To highlight this modification, the variable symbols are indexed with the $\star$ symbol $(E_\star, L_{B\star}, sgk_\star)$. The variable r in B's state is replaced by the already

existing variable r_B that is maintained by the game. These replacements are denoted by a "$\star$" at the right margin of the corresponding lines. We can use r_B instead of r because both variables are incremented equally. Additionally, the counters es_A, er_B are introduced for counting the send and receive operations within an epoch. Hence both counters are reset for every new epoch.

Variable of A	Explanation	Corresponding Variable of B	Explanation
$E^{\lhd}$ in st_A	Epoch to which A sent last ciphertext	$E_\star^{\lhd}$	Epoch for which B received last ciphertext
$E^{\rhd}$ in st_A	Newest epoch that was proposed to A	$E_\star^{\rhd}$	Newest epoch that B proposed to A
es_A	Number of sent ciphertexts to the current epoch $E^{\lhd}$	er_B	Number of received ciphertexts for the oldest cached epoch $E_\star^{\lhd}$
s_A	Total number of sent ciphertexts	r_B	Total number of received ciphertexts

Table 3.1: Variables for indexing arrays in the simulation of games $G_{\geq 2}$. The following conditions hold in presence of a passive adversary: $E_\star^{\lhd} \leq E^{\lhd} \leq E^{\rhd} \leq E_\star^{\rhd}$, $E^{\lhd} = E_\star^{\lhd} \Rightarrow es_A \geq er_B$.

Indexing arrays The index of the arrays $SK_\star, KT_\star$, and the set XSK consists of four parameters: 1) the epoch counter, 2) *within this epoch*, the number of send or receive operations (specified by es_A or er_B), 3) the number of updates (also called *level*) of the secret key via the kuKEM up algorithm (set to 0 for array $KT_\star$), and 4) a tag that indicates whether the entry was used by B's simulation. We emphasize that the index scheme of the key array K instead consists of the epoch counter and the number of *total* send or receive operations (specified by s_A or r_B). It is easy to see that there exists a function that translates these two indexing schemes into each other when disregarding the level and tag parameter. While the indexing via total operation counters simplifies the phrasing of the adversary's winning conditions in our game description, indexing via counters within epochs is more

natural for the simulation in our proof. Table 3.1 provides an intuitive explanation of the index parameters for the simulation.

The indexing scheme for the arrays is constructed to model the usage of secret keys in SRKE—thereby it can also be used to index the symmetric key array $KT_\star$. The index for a freshly generated secret key for a new epoch is obtained by increasing the epoch counter and resetting all remaining index parameters (see Figure 3.17 lines 36, 48, 49 and Figure 3.18 lines 03, 61, 72). When deriving a secret key from the random oracle, the counter within the epoch is increased and the level parameter, to be explained hereafter, is reset (see Figure 3.18 lines 30, 94, 96). Finally, secret keys can be updated. Therefore, the index includes the level parameter that is increased with every update operation (see Figure 3.17 lines 83–86). In addition to the epoch number, the counter within the epoch, and the level of a secret key, a tag $\mathtt{U} \in \{\mathtt{S}, \mathtt{R}\}$ indexes $SK_\star, KT_\star, XSK$. As in the URKE proof, $\mathtt{S}$ denotes values that are used for simulation of A. Entries marked with $\mathtt{R}$ are used by the explicit simulation of B—consequently these entries are only used from the moment of receiving the ciphertext that causes B to become out of sync onwards (see Figure 3.18 lines 53–54).

Usage and computation of arrays As in the URKE proof, the arrays $SK_\star, KT_\star$ store the secret keys and tuples of chaining key, MAC key, and transcript after each sending and receiving operation respectively. Writing and reading the array $KT_\star$ is conducted straight forward as it is in the URKE proof.

Freshly generated secret keys and secret keys as output of the random oracle are directly stored in $SK_\star$. To reassemble the construction's array SK at exposures, or for simulating the receiving out of sync, we use the array of secret keys $SK_\star$ in procedure GetSK (see Figure 3.17). As for the original array SK, this procedure sets the current epoch's secret key in the entry for the current epoch, and all subsequent entries are filled with the first secret key within the respective epoch. To derive the correct updated secret keys for cached epochs in SK (epoch index greater than $E_\star^{|<}$), updates in sync use the

L_A array of A's state to obtain the correct ciphertexts and associated data as update parameters. Out of sync, the helper-array P is first filled with the entries of L_A and then further filled by the simulation of B (see Figure 3.17 line 75, Figure 3.18 lines 53,66,99,100). To correctly simulate the updates, already defined secret keys are taken and further updates are based on the latest existing updated secret key[15] (see Figure 3.17 lines 80 ff.). To adopt secret keys marked with `S` when B becomes out of sync, GetSK copies the most often updated entry (highest level) of each secret key and potentially further updates these keys accordingly. To track when a secret key was initially generated, the counter s is stored for each secret key sent by B (see Figure 3.17 lines 48,49). The number of necessary updates can then be derived by the difference between the number of received ciphertexts by Bob and the value of the send counter attached to the secret key in $SK_\star$.

To ensure that all secret keys are updated according to the construction of the receive algorithm, such that validation in the random oracle is correct, GetSK is invoked a second time at the end of B's receive operation.

For the next game hop, we prepare a cumulative index I that refers to each secret key, used during decapsualtion when B receives (see Figure 3.18 line 81), or to each corresponding public key, used during encapsulation when A sends (see Figure 3.18 line 10). The according single indexes i of these secret keys and their public counterparts used in a send or receive operation, respectively, are concatenated in this index list I. This cumulative index I, hence, points to all secret keys with which input tuples (c, k) to the following random oracle query in a send or receive operation can be decapsulated.

We recall that the third parameter of indexes i indicates the update-level (i.e., the number of updates) of the respective secret key. Array Γ, therefore, stores the number of updates for each secret key. While for the simulation of A Γ is filled at receiving a new public key (see

[15] Please note that the update algorithm is a one way function and regarding the reduction, the simulation has to comply with the KUOW game's update oracle which can only be called sequentially.

Figure 3.17 line 69), the simulation of B can compute Γ along the computation of the secret key in GetSK (see Figure 3.17 line 88). Note that only the first secret key within an epoch needs to be tracked because later keys will not be updated in our SRKE construction.

Exposure and impersonation For tracing exposed secret keys, we introduce the set XSK that is filled with the indexes of exposed secret keys (secret keys that belong to A's public keys). Secret keys are marked to be exposed if they are stored in the exposed array SK. Additionally all subsequent secret keys that are directly derived from these secret keys (i.e., generated within the same epoch) are marked to be exposed, since they can be obtained from the random oracle with the exposed secret keys and the chaining key that is also exposed, or from the publicly known update parameter (ciphertext and associated data). If A and B are out of sync, the updates of secret keys via kuKEM update or random oracle are computed with different values ($ad \,\|\, C$ of A and B are different).[16] Hence, none of A's public keys' secret keys are exposed by B out of sync.

If B is impersonated towards A—i.e., the in-sync state of B for computing the ciphertext to A was exposed and the ciphertext-associated-data pair received by A was not sent by B—, then flag ims is set (see Figure 3.17 line 57). Thereby all future computations by A are traceable and known by the adversary. After the flag ims is set, we stop to store secret keys from the random oracle outputs in A's simulation. Secret keys generated by A after the impersonation are never used by B (because from this moment on the parties are out of sync) and A only needs the respective public keys anyway.

As in URKE, the array CK is introduced in game G_2 to track the triple (c, k, i) after each encapsulation.

Apart from preparations for subsequent games (e.g., setting up indexes, XSK, and ims), this game hop substitutes B's state by introducing global variables and counters. Thereby B is only explicitly

[16] We make sure that no collision in the transcript occurs by aborting on colliding collision keys in the next game hop.

simulated for sending and for computations out of sync. All remaining computations are either conducted by A's simulation or by the helper procedure GetSK.

Game 3 – Internal access to random oracle

Game G_3 is also directly adapted from the URKE proof: the simulation requests the random oracle without providing the input key k^*. There are only two minor differences that do not affect the underlying principle. Firstly, the input to the random oracle is a vector of keys $k^* = k_1 \| \ .. \ \| k_n$. The input transcript t is accordingly also composed of the concatenation of ciphertext vectors $c_1 \| \ .. \ \| c_n$ and the index provided to the internal procedure G consists of multiple secret key indexes $I = i_1 \| \ .. \ \| i_n$. Secondly, not all internal random oracle requests of A are issued via the internal procedure G.

Oracle $\mathrm{H}(k.c, k^*, t)$ $\mathbf{G}_{\geq 0}$
00 $k_1 \| \ .. \ \| k_n \leftarrow k^*$
01 $t' \| \ ad \| \ e \| \ ck \| \ c_1 \| \ .. \ \| \ c_n \leftarrow t$
02 $(k.o, k.c^*, k.m) \leftarrow_{\$} \mathcal{K}^3$; $sk \leftarrow_{\$} \mathcal{SK}$
03 $I \leftarrow \epsilon$; $sen \leftarrow \mathtt{F}$
04 If $\exists I : \mathrm{R}[k.c, t, k, I] \neq \perp$:
05 $(k.o, k.c^*, k.m, sk, sen) \leftarrow \mathrm{R}[k.c, t, k, I]$
06 Else if $\exists I = i_1 \| \ .. \ \| \ i_n : \mathrm{R}[k.c, t, \epsilon, I] \neq \perp$
$\wedge \forall j, 1 \leq j \leq n : i_j \in \mathbb{N}^3 \times [\mathtt{R}]$
$\wedge \mathrm{dec}(sk_j, c_j) = k_j, (sk_j, s) \leftarrow SK_\star[i_j]$: $\mathbf{G}_{\geq 3}$
07 $(k.o, k.c^*, k.m, sk, sen) \leftarrow \mathrm{R}[k.c, t, \epsilon, I]$ $\mathbf{G}_{\geq 3}$
08 Else if $\exists I = i_1 \| \ .. \ \| \ i_n : \mathrm{R}[k.c, t, \epsilon, I] \neq \perp$
$\wedge \forall j, 1 \leq j \leq n : i_j \in \mathbb{N}^3 \times [\mathtt{S}]$
$\wedge CK[c_j, i_j] = k_j$: $\mathbf{G}_{\geq 3}$
09 $(k.o, k.c^*, k.m, sk, sen) \leftarrow \mathrm{R}[k.c, t, \epsilon, I]$ $\mathbf{G}_{\geq 3}$
10 $(e_j, \_\_) \leftarrow i_j$ $\mathbf{G}_{\geq 5}$
11 If $e_n < oos_A \wedge i_n \notin XSK$: Abort $\mathbf{G}_{\geq 5.1}$
12 If $e_1 < oos_A \wedge e_n \geq oos_A$: Abort $\mathbf{G}_{\geq 5.2}$
13 If $e_1 \geq oos_A$: Abort $\mathbf{G}_{\geq 5.3}$
14 $(sk, \_) \leftarrow SK_\star[i_n + 1]$ $\mathbf{G}_{\geq 4}$
15 $(\_, k.m, \_) \leftarrow KT_\star[i_n + 1]$ $\mathbf{G}_{\geq 4}$
16 $\mathrm{R}[k.c, t, k, I] \leftarrow (k.o, k.c^*, k.m, sk, sen)$
17 Return $k.o \| \ k.c^* \| \ k.m \| \ sk$

Proc $\mathrm{G}(k.c, t, I, sen)$ $\mathbf{G}_{\geq 3}$
18 $t' \| \ ad \| \ e \| \ ck \| \ c_1 \| \ .. \ \| \ c_n \leftarrow t$
19 $i_1 \| \ .. \ \| \ i_n \leftarrow I$
20 $k^* \leftarrow \epsilon$
21 If $sen \wedge \exists k^* : \mathrm{R}[k.c, t, k^*, \epsilon] \neq \perp$:
22 Abort
23 Else if $\neg sen \wedge \exists k^* = k_1 \| \ .. \ \| \ k_n$:
$\mathrm{R}[k.c, t, k^*, \epsilon] \neq \perp \wedge \forall j, 1 \leq j \leq n :$
$\mathrm{dec}(sk_j, c_j) = k_j, (sk_j, s) \leftarrow SK_\star[i_j]$:
24 $(k.o, k.c^*, k.m, sk, sen') \leftarrow \mathrm{R}[k.c, t, k^*, \epsilon]$
25 Else:
26 $(k.o, k.c^*) \leftarrow_{\$} \mathcal{K}^2$; $k.m \leftarrow \epsilon$; $sk \leftarrow \epsilon$
27 If $\neg sen$: $k.m \leftarrow_{\$} \mathcal{K}$; $sk \leftarrow_{\$} \mathcal{SK}$
28 $\mathrm{R}[k.c, t, k^*, I] \leftarrow (k.o, k.c^*, k.m, sk, sen)$
29 Return $k.o \| \ k.c^* \| \ k.m \| \ sk$

Proc $\mathrm{SetO}(k.c, t, I, k.m, sk)$ $\mathbf{G}_{\geq 3}$
30 $(k.o, k.c^*, k.m', sk', sen) \leftarrow \mathrm{R}[k.c, t, \epsilon, I]$
31 $\mathrm{R}[k.c, t, \epsilon, I] \leftarrow (k.o, k.c^*, k.m, sk, sen)$
32 Return

Figure 3.19: Random oracle description for proof of SRKE.

In case an impersonation of B towards A was performed by the

adversary such that *ims* is set, the simulation of A strictly follows the construction description and no (kuKEM) challenges are embedded by the simulation. This is sufficient since no future established session key will be challengeable. Consequently the respective random oracle requests do not need to be issued without the knowledge of the kuKEM keys in k^* and, as such, can be computed by using H instead of G.

As described in the previous game, the cumulative index I contains a vector of all indexes to which the send or receive operations encapsulated or decapsulated right before the random oracle invocation, respectively. Thereby the tuple $(C, I, k^*) = (c_1 \parallel .. \parallel c_n, i_1 \parallel .. \parallel i_n, k_1 \parallel .. \parallel k_n)$ is processed in SRKE instead of a tuple consisting of one value each in URKE. The validation of the inputs to the random oracle for finding existing equal entries now works accordingly: If there exist an entry created by an external query for the internal query by B's simulation with the same transcript t and each ciphertexts c_j at the end of t can be decapsulated with the respective secret key with index i_j in I to the key k_j of the external entry's input k^*, then the queries were equal and the output of the internal query is copied from the external one. For queries of A's simulation that collide with an entry made externally (see Figure 3.19 line 22), and for ciphertexts sent by A or B that were equally received by the respective counterpart *before* (see Figure 3.17 line 43 and Figure 3.18 line 24), the game aborts as in the URKE proof.

If the adversary externally requests the random oracle, the validation is again split (in order to comply with the KUOW game's oracles). The vector of ciphertexts in the transcript and the vector of keys in the input key k^* are validated with respect to the existing internally made entries and their vectors of secret key indexes. If there exists an entry made internally (marked with ϵ at the index position of k^*) such that 1) either each secret key i_j from the cumulative index I of this entry can be used to decapsulate the respective ciphertext to the respective key in k^* (see line 06) or 2) all tuples (c_j, i_j, k_j) were stored in the array CK (see line 08), then the external query and internal entry are equal. In this case the output of the external random oracle query is copied from this internal entry.

Splitting the validation is done to prepare the reduction to the KUOW game: The decryption for entries made by B's simulation can be conducted by using the Check oracle. The Solve oracle is modeled by array CK. As in URKE, the reduction will fill CK for exposed secret keys explicitly because the Solve oracle can only be used for unexposed secret keys. In addition to that, the reduction will fill CK also directly for public keys for which the simulation never has access to the respective secret keys. This is the case for all public keys received by A after becoming out of sync without an impersonation (*ims* was not set). The adversary can instead impersonate B *delayed*: if first A is impersonated towards B and then B is exposed, B's signing key is still valid such that the adversary can use it to send valid own ciphertexts to A. Thereby the adversary can send public keys to A for which the adversary (but not the simulation) knows the secret keys. In this case, CK is filled instantly after the encapsulation.

As in the URKE proof, the output MAC key and secret key are sampled independent of the random oracle for internal queries by A.

The probability of the abortion for predefined random oracle entries colliding with internal queries by A, or sent ciphertexts colliding with previously received ciphertexts, is again the bound for distinguishing between games G_2 and G_3, where q_{H} is the number of random oracle requests:

$$\mathrm{Adv}^{G_2,G_3}(\mathcal{D}) \leq \frac{q_{\mathrm{H}} + 2}{|\mathcal{K}|}$$

Requesting the random oracle without providing the cumulated key k^* allows to disregard the decapsulation and hence the usage of secret keys for the simulation of B. For comparing the decapsulation output with the $\perp$ symbol, KUOW's Check oracle will be used such that the secret key of B does not need to be used explicitly. To emphasize this, the input to the decapsulation in line 79 is the respective element from the global secret key array $SK_\star$, implicitly disregarding its second value s.

Game 4 – Random oracle with lazy sampling

In game G_4 we stop to set the output MAC keys and secret keys for random oracle queries by A's simulation if *ims* is not set. Thereby the explicit usage of these keys is shifted to the validation in the random oracle, providing the output for external random oracle requests, and to exposures of B. We will show that the former and letter use cases can be simulated by the reduction to the KUOW or SUF game respectively. The second use case will either not occur without breaking the underlying hardness assumption, or it can also be simulated by using the oracles of the respective games in the reduction.

Please note that our variant of incrementing index i is that the counter es within the epoch is incremented and the level parameter is reset: $i + 1 = (e, es, l, \mathtt{U}) + 1 = (e, es + 1, 0, \mathtt{U})$.

Game 5 – Abortions

Our abortion conditions for SRKE split the key establishment into four cases: 1) keys established by A in sync, 2) the first key established by A out of sync, 3) all remaining keys established by A out of sync, and 4) keys established out of sync by B.

We first want to provide an intuition for these cases: At the beginning of the communication, the parties are in sync. Thereby only an abortion according to case 1) can occur. Manipulating a ciphertext from A to B now introduces cases 4) and, with a delay until the next ciphertext is delivered in the opposite direction, also cases 2) and 3). It is necessary to understand that for provoking an abortion in games $G_{5.2}$ and $G_{5.3}$, a ciphertext to A can only be manipulated after B was already out of sync. The reason for this lies in the abortions of game G_1 and G_3. If G_1 and G_3 do not abort, for ciphertexts to A either an impersonation occurred, or an invalid ciphertext was received by A, or the ciphertext of B was correctly delivered to A. The former case prevents the simulation from embedding challenges to the random oracle, the latter case does not drift A's state out of sync, and the second case provokes the state of A to be erased because the

signature verification will fail.

We split the first three abortion rules for didactic reasons. Therefore, the reduction loses the factor 3 with respect to the advantage in winning the KUOW game. It will become obvious that all three abortions can be summarized to one condition that can be reduced to one instance of the KUOW game such that the reduction would be tight with respect to the employed assumptions.

In the subsequent paragraphs it is called *explicit use* of a value, if the simulation provides this value to the adversary. If the value was only used for a computation—which can be simulated by the underlying game's oracles in the reduction—it was *not explicitly used*.

Game 5.1 – Keys established by A in sync Aborting in game $G_{5.1}$ depends on two conditions for an external random oracle query for which an internally defined entry of A exists: A was in sync when requesting the random oracle for the entry that is externally requested and the secret key sk_n with index i_n, that can be used to decapsulate the last key k_n from the last c_n in t, as input to the random oracle, was not exposed. The public key to the initial secret key of the same epoch as sk_n—secret key with the same value in the epoch index parameter but send and level parameter set to 0—was originally sent by B in sync and correctly delivered to A because otherwise A would have been out of sync already (which would violate the first condition). By condition two, the secret key sk_n was not exposed. This condition can be fulfilled in two ways: either B derived the same secret key sk_n as A, or B became out of sync before an exposure and thereby derived different secret keys in this epoch. In the first case, the condition simply holds because otherwise the secret key would be marked to be exposed (and thereby key k^* would not be challengeable). In the second case, the secret keys of B are differently updated or freshly generated in the random oracle such that the exposure of B's secret keys has no influence on A's established keys. Thereby the reduction can make use of the KUOW game's oracles Up_R and Gen—and if needed Expose—for simulating B's updates and outputs of the

random oracle for the differently derived secret keys.

Apart from obtaining the secret key via an exposure of B, the adversary can obtain it from the output of the random oracle if $n = 1$ holds (i.e., $sk_n = sk_1$ was derived from the random oracle). This, however, can be excluded for the following reason: for this previous internal random oracle entry, outputting secret key with index i_1, there exists a secret key with index i'_n. Both indexes have the same epoch parameters since i'_n was the last secret key index of the random oracle entry outputting secret key with index i_1. As a consequence, the same conditions regarding an abortion in this game for an external query of the random oracle hold. If the adversary requested the random oracle for this entry, outputting the secret key with index i_1, the game would have been aborted before.

Since with providing the correct k_n for c_n and secret key with index i_n, the adversary solves the challenge of the KUOW game, and, as described above, the secret key with i_n was not explicitly used by the simulation, the advantage in distinguishing between G_5 and $G_{5.1}$ can be bounded by the advantage of winning the KUOW game:

$$\mathrm{Adv}^{G_5, G_{5.1}}(\mathcal{D}) \leq \mathrm{Adv}^{\mathrm{kuow}}_{\mathsf{K}}(\mathcal{B}_{\mathsf{K}})$$

Game 5.2 – First key established by A out of sync Game $G_{5.2}$ aborts if the adversary queries the random oracle for the entry that was created by the internal request of A via G directly after becoming out of sync. Thereby at most three types of public keys are used by A for the encapsulations before her internal random oracle request: 1) The public key derived from the last random oracle query's output secret key sk_1, 2) public keys that were sent by B in sync and correctly delivered to A, and 3) public keys received by A that caused her to become out of sync or that were received thereafter. While there exist at least one public key of type 1 and one of type 3, it is not necessary that A also encapsulated to a newly received public key that was sent in sync by B.

As described earlier, for computing challengeable keys, A only becomes out of sync because B drifted out of sync before. Letting A drift

out of sync solely would cause the *ims* flag to be set, or G_1 to abort, or G_3 to abort, or the state of A being erased (preventing the computation of challengeable keys). According to this, B must not have been exposed before drifting out of sync because otherwise the ciphertext, that causes A to become out of sync, is considered as an impersonation such that *ims* is set. As a consequence, the secret key of the newest public key used by A before querying the random oracle, that results from a public key sent by B in sync, was not exposed[17]. This is either the public key to the secret key sk_1 with index i_1 and hence of type 1, or the public key to a secret key sk_j with index $i_j, 1 < j < n_{oos}$ of type 2, where n_{oos} is the (publicly observable) position of the first type-3 public key used for encapsulation in this send operation. Either way the respective secret key (sk_1 or sk_j) was not exposed before B became out of sync and was updated or derived differently from A when B became out of sync afterwards. If this public key is of type 2, its secret key sk_j can only be obtained by an exposure of B because it was freshly generated. Secret key sk_1 of public key of type 1 can also be obtained by the random oracle. The random oracle query that outputs this secret key sk_1 is associated to the previous secret key sk'_n in the same epoch since i'_n was the last secret key index for this previous random oracle entry. If no public key of type 2 exists, this previous secret sk'_n must not have been exposed because otherwise the whole epoch would be marked as exposed which would cause the ciphertext drifting A out of sync to be considered as an impersonation. As such, the random oracle entry outputting the secret key sk_1 fulfills the conditions for an abortion of game $G_{5.1}$ and can thereby be excluded for game $G_{5.2}$.

Conclusively, if there exists a public key of type 2 with secret key index i_j, then this secret key was not explicitly used by the simulation and thereby the adversary's external random oracle with input k_j and c_j can be used to solve the KUOW challenge. Similarly, if no

[17] Results from means that the public key sent in sync by B and received in sync by A was possibly updated and possibly used to feed the random oracle to obtain the public key that was actually used by A in the considered abortion—hence both keys (pk used by A and pk sent by B) are in the same epoch.

kuKEM keys are fed into the internal random oracle query of A that correspond to type-2 public keys, then the secret key with index i_1 was not explicitly used by the simulation and (k_1, c_1) solve the KUOW challenge. Therefore, distinguishing between $G_{5.1}$ and $G_{5.2}$ can be reduced to the KUOW security of the kuKEM:

$$\mathrm{Adv}^{G_{5.1},G_{5.2}}(\mathcal{D}) \leq \mathrm{Adv}^{\mathrm{kuow}}_{\mathsf{K}}(\mathcal{B}_{\mathsf{K}})$$

Game 5.3 – Further keys established by A out of sync In game $G_{5.3}$ for an internally created random oracle entry that causes an abortion, A and B were out of sync when A requested the random oracle for this entry. The first public key, used for the encapsulation before this request, is the result of A's distinct previous random oracle request (meaning that B never queries this request). This previous random oracle request was distinct because by the condition $(e_1 \geq oos_A)$ A requested the random oracle out of sync at least once before, since i_1 has always the same epoch as i'_n from the last random oracle request. Since A and B are out of sync—and thereby their random oracle requests are independent—, the secret key to each first public key for the encapsulation during a send operation can only be obtained by the random oracle. Requesting the random oracle for one of these entries externally would cause the game to abort—either according to game $G_{5.2}$ for the first random oracle query out of sync, or according to this game for all subsequent queries of A. Therefore, by the conditions of the abortion, the secret key to the first public key of each send operation of A out of sync was not explicitly used and hence not known to the adversary such that the tuple (k_1, c_1) solves the KUOW challenge of the kuKEM. Please note that impersonations of B towards A are implicitly excluded since A's simulation invokes G only if *ims* was not set. Hence we can bound the probability of an abortion in $G_{5.3}$ by the advantage of winning the KUOW game:

$$\mathrm{Adv}^{G_{5.2},G_{5.3}}(\mathcal{D}) \leq \mathrm{Adv}^{\mathrm{kuow}}_{\mathsf{K}}(\mathcal{B}_{\mathsf{K}})$$

We note that the abortion event of this game hop can be reduced to the OW security of the KEM solely because the first public key in

A's state is only freshly generated as output of the random oracle but never updated.

Game 5.4 – Keys established by B out of sync The abortion rule of game $G_{5.4}$ of SRKE resembles the one of game $G_{5.2}$ of URKE. For aborting the game, A must not be exposed for the last state in sync between A and B, and B's oldest cached secret key at receiving the ciphertext drifting him out of sync must not have been exposed, either. These conditions imply that an external random oracle query to the entry that defines the last common state values (e.g., $k.m$) would cause an abortion of $G_{5.1}$. Consequently, the MAC key was neither exposed by A, nor by B, nor provided to the adversary via an external random oracle query. Manipulating a ciphertext and delivering it to B without erasing B's state implies that the adversary forged the MAC tag. Hence, the abortion of $G_{5.4}$ can be reduced to winning the SUF game with respect to the MAC:

$$\mathrm{Adv}^{G_{5.3},G_{5.4}}(\mathcal{D}) \leq \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}})$$

To conclude the proof, it is necessary to show that the abortions reduce all random oracle queries by the adversary that output challengeable keys to one of the employed hardness assumptions. Equivalently, we show that the game aborts for the queries that do not output known (and traceable) keys.

According to the transformability between the index scheme in the $\mathrm{KIND}_{\mathsf{R}}$ game and the index scheme for the simulation, the union of A's adversarially known keys KN_A and the union of exposed secret key indexes XSK at an exposure of B are equivalent (see Figure 3.18 lines 40, 42, 43). Since $G_{5.1}$ aborts for all keys established by A in sync for which i_n is not exposed, no challengeable key of A in sync can be obtained from the random oracle. Please note that if $i_n \in XSK$ holds, it is implied that $\forall 1 \leq j \leq n : i_j \in XSK$ because all secret keys are stored in the same state of B.

When drifting out of sync, the $\mathrm{KIND}_{\mathsf{R}}$ game does not increase epochs of the respective party anymore. Consequently impersonations

cause all future keys to be known by the adversary. Equivalently the proof does not embed challenges into the random oracle if B was impersonated towards A and does not consider a valid MAC for a manipulated ciphertext to B as a forgery if A was exposed right before. A forgery is not regarded as such if the oldest epoch in B's state was exposed, either—which is equivalent to the absence of epoch increasing when drifting out of sync.

Since the abortions of games $G_{5.1} - G_{5.3}$ cover all possible internal random oracle queries of A and $G_{5.4}$ excludes the establishment of challengeable keys by B, the adversary cannot derive a challengeable key without letting the game abort. Hence the advantage in winning the game $G_{5.4}$ is 0.

$$\mathrm{Adv}^{G_{5.4}}(\mathcal{A}) = 0$$

Proof result

Summing up the loss due to the game hops described above, provides us with the advantage of an adversary in winning the SRKE $\mathrm{KIND}_{\mathsf{R}}$ game depending on the advantages of adversaries $\mathcal{B}_{\mathsf{K}}$, $\mathcal{B}_{\mathsf{S}}$, and $\mathcal{B}_{\mathsf{M}}$:

$$\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{R}}(\mathcal{A}) \leq 3\mathrm{Adv}^{\mathrm{kuow}}_{\mathsf{K}}(\mathcal{B}_{\mathsf{K}}) + R\mathrm{Adv}^{\mathrm{suf}}_{\mathsf{S}}(\mathcal{B}_{\mathsf{S}}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}}) + \frac{q_{\mathrm{H}} + 2}{|\mathcal{K}|}$$

3.12 Proof of BRKE

We give proof for the security of our generic BRKE construction in Figure 3.13. This proof is described below and depicted in Figure 3.20. Our proof first reduces the creation of signature forgeries to the security of the used signature scheme. Then we show that the adversary cannot win the BRKE $\mathrm{KIND}_{\mathsf{BR}}$ game without breaking the underlying SRKE scheme's security.

Game 1 – Excluding signature forgeries

Similarly to the SRKE proof, game G_1 aborts on signature forgeries that drift the parties' states out of sync. This event can be reduced

to winning the SUF game against the one-time signature scheme.

The major difference between game G_1 in SRKE and BRKE lies within the abortion conditions: In SRKE, impersonations cannot entail forgeries by definition because the respective signing key is leaked to the adversary during the exposure of the impersonated party. The signing key in BRKE can never be exposed to the adversary because it is only used temporarily during the computations in the send algorithm. As such, in BRKE also a ciphertext received as impersonation can contain a signature forgery.

If the adversary injects a manipulated ciphertext that causes the receiving party to drift out of sync, that contains the original verification key, and that is valid with respect to the signature verification, game G_1 aborts. If the respective sender was out of sync before (i.e., if the adversary already injected a manipulated ciphertext in the opposite communication direction before), then this ciphertext does not *cause* the parties to drift out of sync, but only propagates the asynchronicity. This latter case is not considered in the detection of signature forgeries because the underlying SRKE schemes already handle the receipt of manipulated SRKE ciphertexts and the propagation of out-of-sync states.

An adversary distinguishing between the original $\mathrm{KIND}_{\mathsf{BR}}$ game and game G_1 can be used to win the SUF game against the signature scheme. The reduction replaces the singing and verification algorithms with the SUF game's oracles for the ciphertext that entails the first forgery. Hence, the advantage in distinguishing between $\mathrm{KIND}_{\mathsf{BR}}$ and G_1 can be bounded by:

$$\mathrm{Adv}^{\mathrm{kind},G_1}_{\mathsf{BR}}(\mathcal{D}) \leq \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{S}}(\mathcal{B}_{\mathsf{S}})$$

After game G_1 the following invariant holds: If a manipulated ciphertext is received by one of the parties, then either this party's state is erased (due to an invalid signature), or both SRKE receiver states are affected by the manipulation (due to an impersonation)[18]. The

[18]Note that a ciphertext collision in a BRKE ciphertext (prevented by random 'collision' keys in our SRKE scheme from Figure 3.10) can directly be used as

latter statement is true because the adversary must use an own signing key pair for the ciphertext's signature of which the verification key is used as associated data for both SRKE receive algorithms.

Game 2,3 – Key indistinguishability of SRKE

In games G_2, G_3 we use the key indistinguishability of the underlying SRKE schemes to show that the adversary, breaking BRKE, can be used to break one of the employed SRKE instances.

We replace the BRKE challenge keys by random elements from the key space. In game G_2 we do this for keys established from A to B, then in game G_3 for the counter direction accordingly. By matching the winning conditions of an adversary in game G_2 (and G_3) with the winning conditions in the SRKE $\text{KIND}_{\mathsf{SR}}$ game, one can see that these conditions are identical. Please note that, due to the first game hop to game G_1, manipulations of ciphertexts always affect both SRKE ciphertexts.

As a result, the advantage of an adversary, distinguishing between G_1 and G_2 and between G_2 and G_3, respectively, can be bounded by:

$$\text{Adv}_{\mathsf{BR}}^{G_1,G_2}(\mathcal{D}) \leq \text{Adv}_{\mathsf{SR}}^{\text{kind}}(\mathcal{B}_{\mathsf{SR}}), \quad \text{Adv}_{\mathsf{BR}}^{G_2,G_3}(\mathcal{D}) \leq \text{Adv}_{\mathsf{SR}}^{\text{kind}}(\mathcal{B}_{\mathsf{SR}})$$

Since all challenged keys are sampled uniformly at random from the key space in game G_3, the adversary cannot derive information on bit b and consequently the advantage of an adversary is 0:

$$\text{Adv}_{\mathsf{BR}}^{G_3}(\mathcal{A}) = 0$$

Proof result

Summing up the loss due to the game hops described above, provides us with the advantage of an adversary in winning the BRKE $\text{KIND}_{\mathsf{BR}}$ game depending on the advantages of adversaries $\mathcal{B}_{\mathsf{SR}}$ and $\mathcal{B}_{\mathsf{S}}$:

$$\text{Adv}_{\mathsf{BR}}^{\text{kind}}(\mathcal{A}) \leq 2\text{Adv}_{\mathsf{SR}}^{\text{kind}}(\mathcal{B}_{\mathsf{SR}}) + \text{Adv}_{\mathsf{S}}^{\text{suf}}(\mathcal{B}_{\mathsf{S}})$$

a ciphertext collision in the underlying SRKE ciphertexts to break $\text{KIND}_{\mathsf{SR}}$ of the respective (generic) SRKE scheme. Hence, such collisions do not need to be considered for (and handled by) our BRKE scheme.

Simulation $\mathcal{S}^b_{\mathsf{BR}}(\mathcal{A})$
00 For $u \in \{A, B\}$:
01 $s_u \leftarrow 0$; $r_u \leftarrow 0$
02 $e_u \leftarrow 0$; $\mathrm{EP}_u[\cdot] \leftarrow \bot$
03 $E_u^{\lhd} \leftarrow 0$; $E_u^{\rhd} \leftarrow 0$
04 $\mathrm{AC}_u[\cdot] \leftarrow \bot$; $is_u \leftarrow \mathtt{T}$
05 $\mathrm{K}_u[\cdot] \leftarrow \bot$; $\mathrm{XP}_u \leftarrow \emptyset$
06 $\mathrm{KN}_u \leftarrow \emptyset$; $\mathrm{CH}_u \leftarrow \emptyset$
07 $oos_u \leftarrow \infty$; $forge_u \leftarrow \mathtt{F}$ **G₁**
08 $VK_u[\cdot] \leftarrow \bot$ **G₁**
09 $(st_{A,S}, st_{B,R}) \leftarrow_{\$} \mathrm{init}_{\mathsf{SR}}$
10 $(st_{B,S}, st_{A,R}) \leftarrow_{\$} \mathrm{init}_{\mathsf{SR}}$
11 $st_A \leftarrow (st_{A,S}, st_{A,R})$
12 $st_B \leftarrow (st_{B,S}, st_{B,R})$
13 $b' \leftarrow_{\$} \mathcal{A}$
14 For $u \in \{A, B\}$:
15 Require $\mathrm{KN}_u \cap \mathrm{CH}_u = \emptyset$
16 Stop with b'

Oracle $\mathrm{Snd}(u, ad)$
17 Require $st_u \neq \bot$
18 $(st_1, st_2) \leftarrow st_u$
19 $(sgk, vfk) \leftarrow_{\$} \mathrm{gen}_{\mathsf{S}}$; $ad \xleftarrow{\shortparallel} vfk$
20 $VK_u[s_u] \leftarrow vfk$ **G₁**
21 $(st_1, k.o, c_1) \leftarrow_{\$} \mathrm{snd}_A(st_1, ad)$
22 $(st_2, c_2) \leftarrow_{\$} \mathrm{snd}_B(st_2, ad)$
23 $\sigma \leftarrow_{\$} \mathrm{sgn}(sgk, c_1 \,\|\, c_2)$
24 $c \leftarrow vfk \,\|\, c_1 \,\|\, c_2 \,\|\, \sigma$
25 $st_u \leftarrow (st_1, st_2)$
26 If is_u:
27 $\mathrm{AC}_u[s_u] \leftarrow (ad, c)$
28 $\mathrm{EP}_u[s_u] \leftarrow e_u$
29 $E_u^{\rhd} \leftarrow E_u^{\rhd} + 1$
30 $\mathrm{K}_u[\mathtt{S}, e_u, s_u] \leftarrow k.o$
31 $s_u \leftarrow s_u + 1$
32 Return c

Oracle $\mathrm{Reveal}(u, i)$
as in URKE/SRKE (Fig. 3.6)

Oracle $\mathrm{Rcv}(u, ad, c)$
33 Require $st_u \neq \bot$
34 If $is_u \wedge \mathrm{AC}_{\bar{u}}[r_u] \neq (ad, c)$:
35 $is_u \leftarrow \mathtt{F}$
36 $oos_u \leftarrow s_u$ **G₁**
37 If $oos_{\bar{u}} > r_u$: $forge_u \leftarrow \mathtt{T}$ **G₁**
38 If $r_u \in \mathrm{XP}_{\bar{u}}$:
39 $\mathrm{KN}_u \xleftarrow{\cup} \{\mathtt{S}\} \times \mathbb{N} \times [\, s_u, \ldots\,]$
40 $\mathrm{KN}_u \xleftarrow{\cup} \{\mathtt{R}\} \times \mathbb{N} \times [\, r_u, \ldots\,]$
41 If is_u:
42 $E_u^{\lhd} \leftarrow \mathrm{EP}_{\bar{u}}[r_u]$
43 $e_u \leftarrow e_u + 1$
44 $(st_1, st_2) \leftarrow st_u$
45 $vfk \,\|\, c_1 \,\|\, c_2 \,\|\, \sigma \leftarrow c$; $ad \xleftarrow{\shortparallel} vfk$
46 Require $\mathrm{vfy}_{\mathsf{S}}(vfk, c_1 \,\|\, c_2, \sigma)$
47 If $forge_u \wedge vfk = VK_{\bar{u}}[r_u]$: Abort **G₁**
48 $st_1 \leftarrow \mathrm{rcv}_A(st_1, ad, c_2)$
49 Require $st_1 \neq \bot$
50 $(st_2, k.o) \leftarrow \mathrm{rcv}_B(st_2, ad, c_1)$
51 Require $st_2 \neq \bot$
52 $st_u \leftarrow (st_1, st_2)$
53 If $st_u = \bot$: Return $\bot$
54 If is_u: $k.o \leftarrow \diamond$
55 $\mathrm{K}_u[\mathtt{R}, E_u^{\lhd}, r_u] \leftarrow k.o$
56 $r_u \leftarrow r_u + 1$
57 Return

Oracle $\mathrm{Expose}(u)$
58 $\mathrm{KN}_u \xleftarrow{\cup} \{\mathtt{R}\} \times [E_u^{\lhd} \,..\, E_u^{\rhd}] \times [\, r_u, \ldots\,]$
59 If is_u:
60 $\mathrm{XP}_u \xleftarrow{\cup} \{s_u\}$
61 $\mathrm{KN}_{\bar{u}} \xleftarrow{\cup} \{\mathtt{S}\} \times [E_u^{\lhd} \,..\, E_u^{\rhd}] \times [\, r_u, \ldots\,]$
62 Return st_u

Oracle $\mathrm{Challenge}(u, i)$
63 Require $\mathrm{K}_u[i] \in \mathcal{K}$
64 $k \leftarrow b \,?\, \mathrm{K}_u[i] : \$(\mathcal{K})$
65 If $u = A \wedge i \in \{\mathtt{S}\} \times \mathbb{N}^2$
$\vee u = B \wedge i \in \{\mathtt{R}\} \times \mathbb{N}^2$: **G₂**
66 $k \leftarrow_{\$} \mathcal{K}$ **G₂**
67 If $u = B \wedge i \in \{\mathtt{S}\} \times \mathbb{N}^2$
$\vee u = A \wedge i \in \{\mathtt{R}\} \times \mathbb{N}^2$: **G₃**
68 $k \leftarrow_{\$} \mathcal{K}$ **G₃**
69 $\mathrm{K}_u[i] \leftarrow \diamond$
70 $\mathrm{CH}_u \xleftarrow{\cup} \{i\}$
71 Return k

Figure 3.20: Proof of BRKE scheme from Figure 3.13 in BRKE $\mathrm{KIND}_{\mathsf{BR}}$ game from Figure 3.12.

3.13 Modeling ratcheted key exchange

A common criticism in the key exchange community is that many constructions are proposed with an own model that is then only used once to proof this specific construction's security. The resulting problem of different models for many schemes of similar nature is that comparability and comprehensibility of schemes is hampered significantly. We anticipate this by comparing our approach to model security of ratcheted key exchange with prior work on ratcheting and on key agreement in general. Essentially we conclude that our model is in line with prior strategies and with prior notation. Hence, we believe that at our definitions are integrated into the literature. However, ratcheted key exchange is only loosely related to classic key agreement and, therefore, proposing our new models was necessary to meaningfully consider ratcheting.

A security model (for key exchange) mainly consists of three components: 1) communication model with partnering definition, 2) the adversary's ability to obtain information on the communicating parties' secrets, and 3) a winning condition for the security game defined by excluding trivial attacks.

In our definitions (see figures 3.6, 3.9, and 3.12) we combine all three parts of the model in one figure, respectively. The communication model is implicitly given by the oracles $\mathrm{Snd}, \mathrm{Rcv}$. The partnering is defined via the *is* bit (please note that the definition is related to *matching conversations*). The remaining oracles ($\mathrm{Reveal}, \mathrm{Expose}$) define the adversary's ability to obtain secrets from the communicating parties. Finally, the challenge oracle together with the described excluded trivial attacks define the winning conditions for the adversary. Note that excluding trivial attacks within the oracles is in principal equivalent to defining a freshness condition separately. By combining all components of the model in a single compact game definition, the dependencies among them become visible. This especially plays a role in our model since, in contrast to classic key agreement models, our model allows and is based on concurrency in communication (which e.g., influences trivial attacks).

Please note that there is an important difference between ratcheted key exchange and classic key agreement: while key agreement protocols aim to provide the initialization of a communication, ratcheted key exchange serves as a primitive that provides an already initialized session with continuously updated session keys. Both worlds (classic key agreement for initialization and ratcheted key exchange for serving an initialized session) can be composed by using the key, derived from the classic key agreement, to initialize the local session states for the ratcheted key exchange. As such, ratcheting sessions are independent of each other as long as the initializing key agreement provides independent session keys to independent sessions. Hence, our model does not need to consider an environment with multiple users (and multiple sessions each). Consequently users and *long-term keys* do not play a role in ratcheted key exchange.[19]

Both, by defining the security model within one compact game definition, and by disregarding the explicit communication initialization, we are in line with the approach of Bellare et al. [BSJ+17].

In contrast, for example Cohn-Gordon et al. [CCD+17] provide a model that presents the three previously named components (communication model, exposure of secrets, and winning condition) step by step. The significant disadvantage of this approach is that readers must compile these components themselves ad hoc in order to understand the overall security definition (and its guarantees). Besides, we believe that the choice of notation is a matter of taste and in our case, one compact game description seems more appropriate.

As described before, our technique for deriving a model for ratcheted key exchange differs from previous work on ratcheting crucially in our natural consideration of trivial attacks and the resulting unambiguous, objective, strong security definition. One could derive weaker notions of security by restricting the communication model or the adversary's access to the exposure oracles. These weaker notions could be comparable to earlier modeling approaches and would allow

[19] Please note that the construction of Bellare et al. [BSJ+17] does not suffice our model *because* it employs a long-term key for Bob.

for more efficient protocols.

4

Necessity of Strong Building Blocks for Optimally Secure Ratcheting

Contents

4.1 Introduction 122
4.2 Sufficient Security for Key-Updatable KEM 130
4.3 Unidirectional RKE under Randomness Manipulation 138
4.4 kuKEM* to URKE 144
4.5 URKE to kuKEM* 149
4.6 Discussion 161

In our security notions from Chapter 3 we study ratcheting as a primitive from a theoretic point of view, pursuing the strongest security of ratcheting one can hope for. To build accordingly secure constructions we utilize strong, yet inefficient key-updatable primitives—based on hierarchical identity based encryption (HIBE). Related work that followed a comparable definitional approach [JS18a] equally relied for their constructions on these primitives. As neither we nor related works formally justified utilizing these building blocks so far, we answer the yet open question in this chapter under which conditions their use is actually *necessary*.

We revisit our strong notions of ratcheted key exchange (RKE) from Chapter 3, and propose a reasonably extended, slightly stronger security definition. In this security definition, both the exposure of the communicating parties' local states *and* the adversary's ability to attack the executions' randomness are considered. While these two

attacks were partially considered in previous work, we are the first to unify them cleanly in a natural game-based notion.

Due to slight (but meaningful) changes towards our notion from the previous chapter to regard attacks against randomness, we are ultimately able to show that, in order to fulfill strong security for RKE, public key cryptography with (independently) updatable key pairs is a necessary building block. Surprisingly, this implication already holds for the restricted unidirectional RKE case, which was previously instantiated with only standard public key cryptography.

Contributions by the Author This entire chapter has almost exclusively been contributed by the author of this thesis. The full version [BRV20b] of the paper that has been published in the proceedings of ASIACRYPT 2020 [BRV20a] contains, in addition to the contents of this chapter, a formal proof of Theorem 4. This proof can be considered a (slight) adaption of the one presented in Section 3.10. Section 4.5 was jointly contributed by the author of this thesis and a co-author from [BRV20a].

4.1 Introduction

In addition to exposures of locally stored state secrets (which is our focus in Chapter 3), randomness for generating new secrets is often also considered vulnerable. This is motivated by numerous attacks in practice against randomness sources (e.g., [HDWH12]), randomness generators (e.g., [YRS+09, CNE+14]), or exposures of random coins (e.g., [RS09]). Most theoretic approaches try to model this threat by allowing an adversary to *reveal* attacked random coins of a protocol execution (as it was also conducted in related work on ratcheting). This, however, assumes that the attacked protocol honestly and uniformly samples its random coins, either from a high-entropy source or using a random oracle, and that these coins are only afterwards leaked to the attacker. In contrast, practically relevant attacks against bad randomness generators or low-entropy sources

(e.g., [HDWH12, YRS+09, CNE+14]) change the distribution from which random coins are sampled. Consequently, this threat is only covered by a security model if considered adversaries cannot only obtain and reveal but also *influence* the execution's (distribution of) random coins. Thus, it is important to consider randomness *manipulation* instead of reveal, if attacks against randomness are regarded practically relevant.

The overall goal of ratcheting protocols is to reduce the effect of any such non-permanent and/or non-fatal attack to a minimum. For example, an ongoing communication under a non-fatal attack should become secure as soon as the adversary ends this attack or countermeasures become effective. Examples for countermeasures are replacing bad randomness generators via software updates, eliminating state exposing viruses, etc. Motivated by this, most widely used messaging apps are equipped with mechanisms to regularly update the local secrets such that only a short time frame of communication is compromised if an adversary was successful due to obtaining local secrets and/or attacking random coins.

Generic Treatment of Ratcheting as a Primitive. In the following we shortly recall, introduce, and review previous modeling approaches for strongly secure (as opposed to purely practical and relatively weakly secure) ratcheting. We thereby abstractly highlight modeling choices that crucially affect the constructions, secure according to these models respectively. Specifically, we indicate why some models can be instantiated with only public key cryptography (PKC)—bypassing our implication result—and others cannot. In Table 4.1 we summarize this overview.

The initial generic work that considers ratcheted key exchange (RKE) as a primitive and defines its syntax, correctness, and security in a yet impractical variant is by Bellare et al. [BSJ+17]. Abstractly, their concept of *unidirectional* ratcheted key exchange (URKE), depicted in the right part of Figure 4.1 and already introduced in Section 3.3, consist of an initialization that provides two session participants A and B with a state that can then be used by them to repeatedly

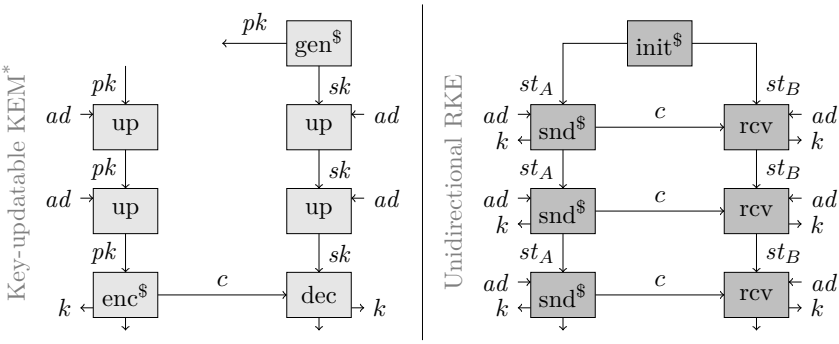

Figure 4.1: Conceptual depiction of kuKEM* and unidirectional RKE. Note the (crucial) difference towards kuKEM (without asterisk), conceptually shown in Figure 3.2: encapsulation and decapsulation of kuKEM* output and may alter the respective input part of the key pair. '$' in the upper index of an algorithm name denotes that the algorithm runs probabilistically and *ad* is associated data.

compute new keys in this session (e.g., for use in higher level protocols). We recall that URKE restricts the communication model such that A is allowed to compute new keys with her state and accordingly send ciphertexts to B who can then compute (the same) keys with his state. During these key computations, A's and B's states are updated, respectively, to minimize the effect of state exposures. We emphasize that B can only comprehend key computations from A, on receipt of a ciphertext, but cannot actively initiate the computation of new keys. Beyond this restriction of the communication model, the security definition by Bellare et al. only allows the adversary to expose A's temporary local state secrets, while B's state cannot be exposed (which in turn requires no forward-secrecy with respect to state updates by B). Following Bellare et al., we propose in Chapter 3 a revised security definition of unidirectional RKE (thereby we also allow the exposure of B's state) and extend the communication model to define syntax, correctness, and security of *sesqui*directional RKE (SRKE: additionally allows B to only send special update ciphertexts to A that do not trigger a new key computation but help him to recover from state exposures) and *bi*directional RKE (BRKE: defines

A and B to participate equivalently in the communication). With a similar instantiation, Jaeger and Stepanovs [JS18a] define security for bidirectional channels under state exposures and randomness reveal.

All of the above mentioned approaches define security *optimally* with respect to their syntax definition and the adversary's access to the primitive execution (modeled via oracles in the security game). This is reached by declaring secrets insecure *iff* the adversary conducted an unpreventable/trivial attack against them (i.e., a successful attack that no instantiation can prevent). Consequently, fixing syntax and oracle definitions, no stronger security definitions exist.

Relaxed Security Notions. Subsequent to these strongly secure ratcheting notions, multiple weaker formal definitions for ratcheting were proposed that consider special properties such as strong explicit authentication [DV19], out of order receipt of ciphertexts [ACD19], or primarily target on allowing efficient instantiations [JMM19, CDV19].

While these works are syntactically similar, we shortly sketch their different relaxations regarding security—making their security notions sub-optimal. Durak and Vaudenay [DV19] and Caforio et al. [CDV19] forbid the adversary to perform impersonation attacks against the communication between A and B during the establishment of a *secure* key. Thus, they do not require recovery from state exposures—which are a part of impersonation attacks—in all possible cases, which we denote as 'partial recovery' (see Table 4.1). Furthermore, both works neglect bad randomness as an attack vector. In the security experiments by Jost et al. [JMM19] and Alwen et al. [ACD19] constructions can delay the recovery from attacks longer than necessary (Jost et al. therefore temporarily forbid the exposure of the local state). Additionally, they do not require the participants' states to become incompatible (immediately) on active attacks against the communication.

Instantiations of Ratcheting. Interestingly, both mentioned *uni*directional RKE instantiations that were defined to depict opti-

[1] '*Unnecessary*' refers to restrictions beyond those that are immediately implied by optimal security definitions (that only restrict the adversary with respect to unpreventable/trivial attacks).

	(a) Interaction	(b) State Exposure	(c) Bad Randomness	(d) Recovery
C+ [CCD+17]	↔	Always allowed	Reveal	Delayed
B+ [BSJ+17]	→	Only allowed for A	Reveal	Immediate
Chapter 3 published in [PR18b, PR18a]	→	Always allowed	Not considered	Immediate
	↦	Always allowed	Not considered	Immediate
	↔	Always allowed	Not considered	Immediate
JS [JS18a]	↔	Always allowed	Reveal	Immediate
DV [DV19]	↔	Always allowed	Not considered	Partial
JMM [JMM19]	→	Partially restricted	Reveal	(Immediate)
	↦	Partially restricted	Reveal	(Immediate)
	↔	Partially restricted	Reveal	(Immediate)
ACD [ACD19]	↔	Always allowed	Manipulation	Delayed
CDV [CDV19]	↔	Always allowed	Not considered	Delayed
This work	→	Always allowed	Manipulation	Immediate

Table 4.1: Differences in security notions of ratcheting regarding (a) uni- (→), sesqui- (↦), and bidirectional (↔) interaction between A and B, (b) when the adversary is allowed to expose A's and B's state (or when this is *unnecessarily* restricted), (c) the adversary's ability to reveal or manipulate algorithm invocations' random coins, and (d) how soon and how complete recovery from these two attacks into a secure state is required of *secure* constructions (or if *unnecessary* delays or exceptions for recovery are permitted).[1] Recovery from attacks required by Jost et al. [JMM19] is *immediate* in so far as their restrictions of state exposures introduce delays implicitly. Gray marked cells indicate the reason (i.e., relaxations in security) why respective instantiations can rely on standard PKC only (circumventing our implication result). Rows without gray marked cells have no construction based on pure PKC.

mal security (i.e., our construction from Section 3.4 and [BSJ+17]) as well as bidirectional real-world examples such as the Signal protocol (analyzed in [CCD+17]), and instantiations of the above named relaxed security notions [DV19, JMM19, ACD19, CDV19] only rely on standard PKC (cf. rows in Table 4.1 with gray cells).

In contrast, both mentioned optimally secure bidirectional ratcheting variants (i.e., sesquidirectional and bidirectional RKE from sections 3.6 and 3.9, and bidirectional strongly secure channel [JS18a]) are based on *key-updatable public key encryption*, which can be built from hierarchical identity based encryption (HIBE). We recall the intuitive concept, formally already introduced in Section 3.2: key-updatable public key encryption is standard public key encryption that additionally allows to update public key and secret key independently with respect to some associated data (a conceptual depiction of

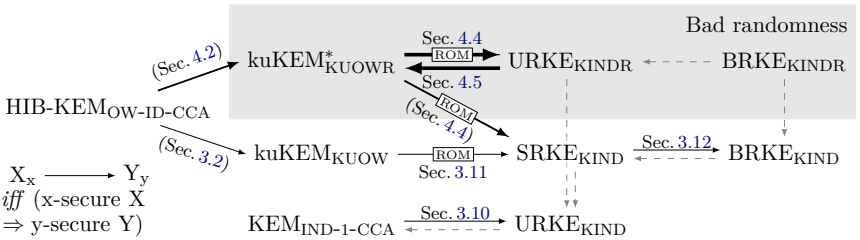

Figure 4.2: The contributions of this chapter (bold arrows) and their connection to results from Chapter 3 (thin arrows) involving RKE (uni-, sesqui-, and bidirectional) and KEM (standard, hierarchical-identity-based, and key-updatable) primitives. ROM indicates that the proof holds in the random oracle model. kuKEM$^{*}_{\text{KUOWR}}$ $\Rightarrow_{\text{ROM}}$ SRKE$_{\text{KIND}}$ is not formally proven in this chapter, but we point out that the proof of kuKEM$_{\text{KUOW}}$ $\Rightarrow_{\text{ROM}}$ SRKE$_{\text{KIND}}$ from Section 3.11 can be rewound. Gray dashed connections indicate trivial implications (due to strictly weaker syntax or security definitions).

this is on the left side of Figure 4.1). Thereby an updated secret key cannot be used to decrypt ciphertexts that were encrypted to previous (or different) versions of this secret key, where versions are defined over the associated data used for updates.

Necessity for Strong Building Blocks. Natural questions that arise from this line of work are, whether and under which conditions such strong (HIBE-like) building blocks are not only sufficient but also necessary to instantiate the strong security of (bidirectional) RKE. In order to answer these questions, we build key-updatable public key cryptography from ratcheted key exchange. Consequently we affirm the necessity and provide (sufficient) conditions for relying on these strong building blocks. We therefore minimally adjust the syntax of key-updatable key encapsulation mechanism (kuKEM) from Section 3.2 and consider the manipulation of algorithm invocations' random coins in our security definitions of kuKEM and RKE.[2]

While, despite these changes of syntax and security towards prior definitions, we prove that RKE can still be built from kuKEM, we

[2]Recall that randomness manipulation was not considered in a security definition that aimed for optimal security in the literature of ratcheting yet (cf. Table 4.1).

also prove that kuKEM can be built from RKE (see Figure 4.2). As a result we show that:

- kuKEM* (with one-way security under manipulation of randomness) $\Rightarrow_{\text{ROM}}$ Unidirectional RKE (with key indistinguishability under manipulation of randomness),
- Unidirectional RKE (with key indistinguishability under manipulation of randomness) $\Rightarrow$ kuKEM* (with one-way security under manipulation of randomness).

The asterisk at kuKEM* indicates the minimal adjustment to the kuKEM syntax definition from Section 3.2.[3]

Given the security notions established in honest randomness setting and their connections to each other, one would also expect Group RKE $\Rightarrow$ Bidirectional RKE $\Rightarrow$ Sesquidirectional RKE $\Rightarrow$ Unidirectional RKE to follow. Hence, our results indicate that stronger RKE variants also likely require building blocks as hard as kuKEM*. Furthermore, we can show that our results from Section 3.6 remain valid under the changed notion of kuKEM*: One-way security under manipulation of randomness of kuKEM* $\Rightarrow_{\text{ROM}}$ Key indistinguishability of *sesquidirectional* RKE.

Interestingly, these results induce that (when considering strong security) ratcheted key exchange requires these strong (HIBE-like) building blocks not only for bidirectional communication settings, but already for the unidirectional case. Both mentioned previous unidirectional RKE schemes can bypass our implication because they forbid exposures of B's state [BSJ+17] or assume secure randomness as in Section 3.3 (see Table 4.1). We describe attacks against each of both constructions in our URKE security definition from this chapter in Section 4.4.1. Since the mentioned relaxed security definitions of ratcheting [CCD+17, DV19, JMM19, ACD19, CDV19] restrict the adversary more than necessary in exposing states, solving (potentially valid) embedded game challenges, manipulating the communication

[3] For the kuKEM* we consider one-way security as it suffices to achieve strong security for RKE. It is obvious that the same results hold for key indistinguishability.

between the session participants, or attacking invocations' random coins (and thus violate either of our security definition's conditions), it remains feasible to instantiate them with standard public key primitives as well (see Table 4.1). Although our analysis is partially motivated by the use of kuKEM in the constructions from Chapter 3 and in [JS18a], we do not ultimately answer whether these particular constructions necessarily relied on it. Rather we provide a clean set of conditions under which RKE and kuKEM clearly imply each other as we do not consider the justification of previous constructions but a clear relation for future work important. (However, we extensively discuss whether and how our approach can be extended accordingly in Section 4.6.)

Thus, we show that sufficient conditions for necessarily relying on kuKEM as a building block of RKE are: (a) unrestricted exposure of both parties' local states, (b) consideration of attacks against algorithm invocations' random coins, and (c) required immediate recovery from these two attacks into a secure state by the security definition (i.e., the adversary is only restricted with respect to unpreventable/trivial attacks).[4]

Contributions. The contributions of this chapter can be summarized as follows:

- We are the first who systematically define optimal security of key-updatable KEM and unidirectional RKE under randomness manipulation (in sections 4.2 and 4.3) and thereby consider this practical threat in addition to state exposures in an instantiation-independent notion of RKE. Thereby we substantially enhance the respective models from Chapter 3.
- In Section 4.4, we construct unidirectional RKE generically from a kuKEM* to show that the latter suffices as a building block

[4]Note that there may exist further sets of sufficient conditions for relying on kuKEMs since, for example, sesqui- and bidirectional RKE from sections 3.5 and 3.8 violate condition (b) but base on kuKEMs as well. We refer the reader to Section 3.7 for a detailed explanation of why these scheme presumably also must rely on a kuKEM. We leave the identification of further sets of conditions as future work.

for the former under manipulation of randomness.
- To show that kuKEM* is not only sufficient but also necessary to build unidirectional RKE (under randomness manipulation), we provide a construction of kuKEM* from a generic unidirectional RKE scheme in Section 4.5.

With our results we distill the core building block of strongly secure ratcheted key exchange down to its syntax and security definition. This allows further research to be directed towards instantiating kuKEM* schemes that are more familiar and easier in terms of security requirements, rather than attempting to construct seemingly more complex RKE primitives.[5] Simultaneously, our results indicate the cryptographic hardness of ratcheted key exchange and thereby help to systematize and comprehend the security definitions and different dimensions of ratcheting in the literature. As a consequence, our results contribute to a fact-based trade-off between security and efficiency for RKE by providing requirements for relying on heavy building blocks and thereby revealing respective bypasses.

4.2 Sufficient Security for Key-Updatable KEM

A key-updatable key encapsulation mechanism (kuKEM) is a key encapsulation mechanism that provides update algorithms for public key and secret key with respect to some associated data respectively. In order to allow for our equivalence result, we minimally adjust the orig-

[5] For example, the bidirectional channel construction in the proceedings version of [JS18a] is not secure according to the security definition (but a corrected version is published as [JS18b]), in the acknowledgments of [PR18a] (i.e., the paper on which Chapter 3 bases) it is mentioned that an early submitted version of our construction was also flawed, and for an earlier version of [DV19] (available as [DV18]) we detected during our work (and informed the authors) that the construction was insecure under bad randomness such that the updated proceedings version disregards attacks against randomness entirely. Finally, we detected and reported that the construction of HkuPke in [JMM19] is not even correct.

inal kuKEM notion from Section 3.2 and call it kuKEM*. The small, yet crucial changes comprise allowed updates of public and secret key during encapsulation and decapsulation (in our syntax definition) as well as the adversary's ability to manipulate utilized randomness of encapsulations (in our security definition). In Section 4.5 the rationales behind these changes are clarified. In order to provide a coherent definition, we not only describe alterations towards Section 3.2 but define kuKEM* entirely (as we consider our changes to be a significant contribution and believe that this strengthens comprehensibility).

Syntax A kuKEM* for a space of encapsulated keys $\mathcal{K}$ is a quadruple $\mathsf{K} = (\mathrm{gen}_{\mathsf{K}}, \mathrm{up}, \mathrm{enc}, \mathrm{dec})$ of algorithms together with spaces of public keys $\mathcal{PK}$ and secret keys $\mathcal{SK}$, a space of associated data $\mathcal{AD}$ for updating the keys, a ciphertext space $\mathcal{C}$ (with $\mathcal{AD} \cap \mathcal{C} = \emptyset$). Furthermore we define $\mathcal{R}$ as the space of random coins used during the encapsulation. In contrast to the syntax of kuKEM in Section 3.2, the encapsulation algorithm of kuKEM* also outputs a (potentially modified version of the input) public key, and decapsulation algorithm accordingly outputs a (potentially modified version of the input) secret key—as a result, the kuKEM* is stateful (where the public key is a public state).[6] A shortcut notation for kuKEM* algorithms is

$$\begin{array}{rcl} & \mathrm{gen}_{\mathsf{K}} \rightarrow_{\$} \mathcal{PK} \times \mathcal{SK} & \mathcal{PK} \times \mathcal{R} \rightarrow \mathrm{enc} \rightarrow \mathcal{PK} \times \mathcal{K} \times \mathcal{C} \text{ or} \\ \mathcal{PK} \times \mathcal{AD} \rightarrow \mathrm{up} \rightarrow \mathcal{PK} & & \mathcal{PK} \rightarrow \mathrm{enc} \rightarrow_{\$} \mathcal{PK} \times \mathcal{K} \times \mathcal{C} \\ \mathcal{SK} \times \mathcal{AD} \rightarrow \mathrm{up} \rightarrow \mathcal{SK} & & \mathcal{SK} \times \mathcal{C} \rightarrow \mathrm{dec} \rightarrow (\mathcal{SK} \times \mathcal{K}) \cup \{(\perp, \perp)\} \end{array}$$

Correctness The correctness for kuKEM* is (for simplicity) defined through game $\mathrm{FUNC}_{\mathsf{K}}$ (see Figure 4.3), in which an adversary $\mathcal{A}$ can query encapsulation, decapsulation, and update oracles. The adversary (against correctness) wins if different keys are computed during decapsulation and the corresponding encapsulation even though com-

[6] As kuKEM* naturally provides no security for encapsulated keys if the adversary can manipulate the randomness for $\mathrm{gen}_{\mathsf{K}}$ already, we only consider the manipulation of random coins for enc.

patible key updates were conducted and ciphertexts from encapsulations were directly forwarded to the decapsulation oracle.

A kuKEM* scheme K is correct if the probability of winning game FUNC_{K} from Figure 4.3 is $\Pr[\text{FUNC}_{\mathsf{K}}(\mathcal{A}) \to 1] = 0$ for every $\mathcal{A}$.

Game $\text{FUNC}_{\mathsf{K}}(\mathcal{A})$
00 $(pk, sk) \leftarrow_{\$} \text{gen}_{\mathsf{K}}$
01 $\text{K}[\cdot] \leftarrow \bot$
02 $trs \leftarrow \epsilon$; $trr \leftarrow \epsilon$
03 Invoke $\mathcal{A}$
04 Stop with 0

Oracle $\text{Up}_S(ad)$
05 Require $ad \in \mathcal{AD}$
06 $pk \leftarrow \text{up}(pk, ad)$
07 $trs \xleftarrow{\shortparallel} ad$
08 Return

Oracle $\text{Up}_R(ad)$
09 Require $ad \in \mathcal{AD}$
10 $sk \leftarrow \text{up}(sk, ad)$
11 $trr \xleftarrow{\shortparallel} ad$
12 Return

Oracle Enc()
13 $(pk, k, c) \leftarrow_{\$} \text{enc}(pk)$
14 $trs \xleftarrow{\shortparallel} c$
15 $\text{K}[trs] \leftarrow k$
16 Return (pk, c)

Oracle $\text{Dec}(c)$
17 Require $c \in \mathcal{C}$
18 $(sk, k) \leftarrow \text{dec}(sk, c)$
19 $trr \xleftarrow{\shortparallel} c$
20 If $trr \preceq trs$:
21 Reward $k \neq \text{K}[trr]$
22 Return

Figure 4.3: The correctness notion of kuKEM* captured through game FUNC.

Security Here we describe KUOWR security of kuKEM* as formally depicted in Figure 4.4. KUOWR defines one-way security of kuKEM* under randomness manipulation in a multi-instance/multi-challenge setting.

Intuitively, the KUOWR game requires that a secret key can only be used for decapsulation of a ciphertext if prior to this decapsulation all updates of this secret key and all decapsulations with this secret key were consistent with the updates of and encapsulations with the respective public key. This is reflected by using the transcript (of public key updates and encapsulations or secret key updates and decapsulations) as a reference to encapsulated 'challenge keys' and secret keys.

In Figure 4.4 we denote changes with respect to KUOW security from Figure 3.3 by adding '·' at the beginning of lines.

In order to let the adversary play with the kuKEM*'s algorithms, the game provides oracles Gen, Up_S, Up_R, Enc, and Dec. Thereby instances (i.e., key pairs) can be generated via oracle Gen and are

referenced in the remaining oracles by a counter that refers to when the respective instance was generated.

Game $\mathrm{KUOWR}_{\mathsf{K}}(\mathcal{A})$
00 $n \leftarrow 0$
01 Invoke $\mathcal{A}$
02 Stop with 0

Oracle Gen
03 $n \leftarrow n+1$
04 $(pk_n, sk_n) \leftarrow_{\$} \mathrm{gen}_{\mathsf{K}}$
05 $\mathrm{CK}_n[\cdot] \leftarrow \bot$; $\mathrm{XP}_n \leftarrow \emptyset$
06 $trs_n \leftarrow \epsilon$; $trr_n \leftarrow \epsilon$
07 $\mathrm{SK}_n[\cdot] \leftarrow \bot$
08 $\mathrm{SK}_n[trr_n] \leftarrow sk_n$
09 Return pk_n

Oracle $\mathrm{Up}_S(i, ad)$
10 Require $1 \le i \le n \wedge ad \in \mathcal{AD}$
11 $pk_i \leftarrow \mathrm{up}(pk_i, ad)$
12 $trs_i \xleftarrow{\shortparallel} ad$
13 Return pk_i

Oracle $\mathrm{Enc}(i, rc)$
14 Require $1 \le i \le n$
15 · Require $rc \in \mathcal{R} \cup \{\epsilon\}$
16 · If $rc = \epsilon$: $mr \leftarrow \mathtt{F}$; $rc \leftarrow_{\$} \mathcal{R}$
17 · Else: $mr \leftarrow \mathtt{T}$
18 · $(pk_i, k, c) \leftarrow \mathrm{enc}(pk_i; rc)$
19 · $trs_i \xleftarrow{\shortparallel} c$
20 · If $mr = \mathtt{F}$: $\mathrm{CK}_i[trs_i] \leftarrow k$
21 · Return (pk_i, c)

Oracle $\mathrm{Solve}(i, tr, k)$
22 Require $1 \le i \le n$
23 Require $tr \notin \mathrm{XP}_i$
24 Require $\mathrm{CK}_i[tr] \neq \bot$
25 Reward $k = \mathrm{CK}_i[tr]$
26 Return

Oracle $\mathrm{Up}_R(i, ad)$
27 Require $1 \le i \le n \wedge ad \in \mathcal{AD}$
28 $sk_i \leftarrow \mathrm{up}(sk_i, ad)$
29 $trr_i \xleftarrow{\shortparallel} ad$
30 $\mathrm{SK}_i[trr_i] \leftarrow sk_i$
31 Return

Oracle $\mathrm{Dec}(i, c)$
32 Require $1 \le i \le n \wedge c \in \mathcal{C}$
33 · $(sk_i, k) \leftarrow \mathrm{dec}(sk_i, c)$
34 · $trr_i \xleftarrow{\shortparallel} c$
35 · $\mathrm{SK}_i[trr_i] \leftarrow sk_i$
36 · If $\mathrm{CK}_i[trr_i] \neq \bot$:
37 · Return
38 · Return k

Oracle $\mathrm{Expose}(i, tr)$
39 Require $1 \le i \le n$
40 · Require $\mathrm{SK}_i[tr] \in \mathcal{SK}$
41 · $\mathrm{XP}_i \xleftarrow{\cup} \{tr^* \in (\mathcal{AD} \cup \mathcal{C})^* : tr \prec tr^*\}$
42 Return $\mathrm{SK}_i[tr]$

Figure 4.4: Security experiment KUOWR, modeling one-way security of kuKEM* in a multi-instance/multi-challenge setting under randomness manipulation. Lines of code tagged with '·' are (substantially) modified with respect to KUOW security from Figure 3.3. Line 41 is a shortcut notion that can be implemented efficiently. CK: challenge keys, XP: exposed secret keys, *trs*, *trr*: transcripts.

For encapsulation via oracle Enc, the adversary can either choose the invocation's random coins by setting rc to some value that is not the empty string ϵ or let the encapsulation be called on fresh randomness by setting $rc = \epsilon$ (line 16). In the former case, the adversary

trivially knows the encapsulated key. Thus, only when called with fresh randomness, the encapsulated key is marked as a challenge key in array CK (line 20).

The variables CK, SK, and XP (the latter two are explained below) are indexed via the transcript of operations on the respective key pair part. As public keys and secret keys can uniquely be referenced via the associated data under which they are updated and via ciphertexts that have been encapsulated or decapsulated by them, the concatenation of these values (i.e., sent or received transcripts trs, trr) are used as references to them in the KUOWR game.

On decapsulation of a key that is not marked as a challenge, the respective key is output to the adversary. Challenge keys are of course not provided to the adversary as thereby the challenge would be trivially solved (line 36).

Via oracle Expose, the adversary can obtain a secret key of specified instance i that results from an operation referenced by transcript tr. As described above, the transcript, to which a secret key refers, is built from the associated data of updates to this secret key (via oracle Up_R) and the ciphertexts of decapsulations with this secret key (via oracle Dec) as these two operations may modify the secret key. As all operations, performed with an exposed secret key, can be traced by the adversary (i.e., updates and decapsulations; note that both are deterministic), all secret keys that can be derived from an exposed secret key are also marked exposed via array XP (line 41).

Finally, an adversary can solve a challenge via oracle Solve by providing a guess for the challenge key that was encapsulated for an instance i with the encapsulation that is referenced by transcript tr. Recall that the transcript, to which an encapsulation refers, is built from the associated data of updates to the respective instance's public key (via oracle Up_S) and the ciphertexts of encapsulations with this instance's public key (via oracle Enc) as these two operations may modify the public key for encapsulation. If the secret key for decapsulating the referenced challenge key is not marked exposed (line 23) and the guess for the challenge key is correct (line 24), then game KUOWR stops with '1' (via 'Reward') meaning that the adversary

wins.

We define the advantage of any adversary $\mathcal{A}$ against a kuKEM* scheme K in game KUOWR from Figure 4.4 as $\mathrm{Adv}^{\mathrm{kuowr}}_{\mathsf{K}}(\mathcal{A}) = \Pr[\mathrm{KUOWR}_{\mathsf{K}}(\mathcal{A}) \to 1]$.

As it suffices to show equivalence with key indistinguishability of RKE (in the ROM), we chose to consider one-way security as opposed to key indistinguishability for kuKEM*.

Differences compared to KUOW Security The main difference in our definition of KUOWR security compared to KUOW security from Section 3.2 is that we allow the adversary to manipulate the execution's random coins here. As we define encapsulation and decapsulation to (potentially) update the used public key or secret key, another conceptual difference is that we only allow the adversary to encapsulate and decapsulate once under each public and secret key. Thus, we assume that public and secret keys are overwritten on encapsulation and decapsulation, respectively. In contrast to our security definition here, in the KUOW security game only the current secret key of an instance can be exposed. Even though we assume the secret key to be replaced by its newer versions on updates or decapsulations, there might be, for example, backups that store older secret key versions. As a result we lift the restriction of only allowing exposures of the current secret key.[7] An important notational choice is that we index the variables with transcripts *trs*, *trr* instead of integer counters. This notation reflects the idea that public key and secret key only stay compatible as long as they are used correspondingly and immediately diverge on different associated data or tampered ciphertexts.

We further highlight the fundamental difference towards HkuPke by Jost et al. [JMM19]. Their notion of HkuPke does not allow (fully adversary-controlled) associated data on public and secret key updates and additionally requires (authenticated) interaction between

[7] It is important to note that the equivalence between KUOWR security of kuKEM* and KINDR security of URKE is independent of this definitional choice—if either both definitions allow or both definitions forbid the exposure of also past secret keys or states respectively, equivalence can be shown.

the holders of the key parts thereby. Looking ahead, this makes this primitive insufficient for diverging the public key from the secret key (in the states) of users A and B during an impersonation of A towards B in (U)RKE (especially under randomness manipulation). This is, however, required in an optimal security definition but explicitly excluded in the sub-optimal RKE notion by Jost et al. [JMM19]. Since the syntax of HkuPke is already inadequate to reflect this security property, we cannot provide a separating attack. Nevertheless, we further expound this weakness in Section 4.4.2.

Instantiation A kuKEM* scheme, secure in the KUOWR game, can be generically constructed from an OW-CCA adaptively secure hierarchical identity based key encapsulation mechanism (HIBE). The construction—almost the same as in Section 3.2—is provided for completeness in Figure 4.5. The update of public keys is the concatenation of associated data (interpreted as identities in the HIBE) and the update of secret keys is the delegation to lower level secret keys in the identity hierarchy. The reduction is immediate: After guessing for which public key and after how many updates the challenge encapsulation that is solved by the adversary is queried, the challenge from the OW-CCA game is embedded into the respective KUOWR challenge.

Sufficiency of KUOWR **for SRKE** Before proving equivalence between security of key-updatable KEM and ratcheted key exchange, we shed a light on implications due to the differences between our notion of kuKEM* and its KUOWR security and the notion of kuKEM and its KUOW security from Section 3.2.

Remark 1 *Even though the* KUOWR *game provides more power to the adversary in comparison to the* KUOW *game by allowing manipulation of random coins, exposures of past secret keys, and providing an explicit decapsulation oracle (instead of an oracle that only allows for checks of ciphertext-key pairs; cf., Figure 3.3), the game also restricts the adversary's power by only allowing decapsulations under the current secret key of an instance (as opposed to also checking ciphertext-*

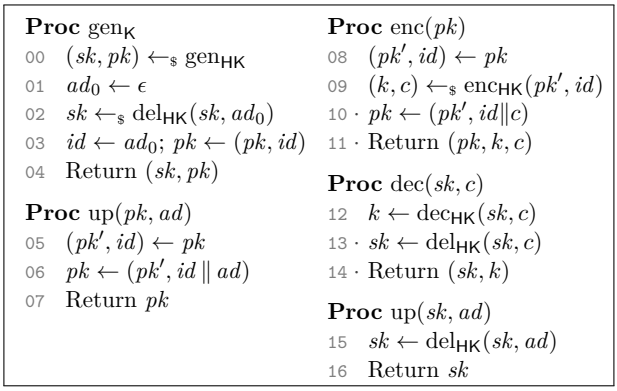

Figure 4.5: Generic construction of kuKEM* from an HIBE scheme. The small changes towards Figure 3.4 (i.e., adding an internal key update in encapsulation and decapsulation, respectively) are marked with a '·'.

key pairs for past secret keys of an instance as in the KUOW *game). One can exploit this and define protocols that are secure with respect to one game definition but allow for attacks in the other game. Consequently, neither of both security definitions implies the other one.*

Despite the above described distinction between both security definitions, KUOWR security suffices to build sesquidirectional RKE according to game KIND from Figure 3.9—which was yet the weakest notion of security of RKE for which a construction was built from a key-updatable public key primitive. The ability to check ciphertext-key pairs under past versions of secret keys of an instance is actually never used in the proof from Section 3.11. The only case in which this Check oracle is used in this proof is B's receipt of a manipulated ciphertext from the adversary. Checking whether a ciphertext-key pair for the current version of a secret key of an instance is valid, can of course be conducted by using the Dec oracle of our KUOWR notion. For full details on this proof we refer the reader to Section 3.11.

Consequently, for the construction of KIND secure sesquidirectional RKE (according to Figure 3.9), the used kuKEM must either be KUOW secure (see Figure 3.3) or KUOWR secure (see Figure 4.4),

which is formally phrased in the following observation. Thus, even though these notions are not equivalent, they both suffice for constructing KIND secure sesquidirectional RKE.

Observation 1 *The sesquidirectional RKE protocol* R *from Figure 3.10 offers* key indistinguishability *according to Figure 3.9 if function* H *is modeled as a random oracle, the* kuKEM* *provides* KUOWR *security according to Figure 4.4, the one-time signature scheme provides* SUF *security according to Figure 2.3, the MAC scheme* M *provides* SUF *security according to Figure 2.2, and the symmetric-key space of the* kuKEM* *is sufficiently large.*

4.3 Unidirectional RKE under Randomness Manipulation

Unidirectional RKE (URKE) is the simplest variant of ratcheted key exchange. After a common initialization of a session between two parties A and B, it enables the continuous establishment of keys within this session. In this unidirectional setting, A can initiate the computation of keys repeatedly. With each computation, a ciphertext is generated that is sent to B, who can then comprehend the computation and output (the same) key. Restricting A and B to this unidirectional communication setting, in which B cannot respond, allows to understand the basic principles of ratcheted key exchange. For the same reasons we provided the whole definition of kuKEM* before (i.e., we consider our changes significant and non-trivial, and we aim for a coherent presentation), we fully define URKE under randomness manipulation below.

Syntax We recall that URKE is a triple $\mathsf{UR} = (\mathrm{init}, \mathrm{snd}, \mathrm{rcv})$ of algorithms defined over spaces of A's and B's states $\mathcal{S}_A$ and $\mathcal{S}_B$, respectively, an associated-data space $\mathcal{AD}$, a ciphertexts space $\mathcal{C}$, and a space of keys $\mathcal{K}$ established between A and B. We extend the syntax of URKE by explicitly regarding the utilized randomness of the

snd algorithm. Consequently we define $\mathcal{R}$ as the space of random coins $rc \in \mathcal{R}$ used in snd. To highlight that A only sends and B only receives in URKE, we may add 'A' and 'B' as handles to the index of snd and rcv, respectively. A shortcut notation for these algorithms is

$$\begin{array}{rcccl} & & \text{init} & \to_{\$} & \mathcal{S}_A \times \mathcal{S}_B \\ \mathcal{S}_A \times \mathcal{AD} \times \mathcal{R} & \to & \text{snd} & \to & \mathcal{S}_A \times \mathcal{K} \times \mathcal{C} \text{ or} \\ \mathcal{S}_A \times \mathcal{AD} & \to & \text{snd} & \to_{\$} & \mathcal{S}_A \times \mathcal{K} \times \mathcal{C} \\ \mathcal{S}_B \times \mathcal{AD} \times \mathcal{C} & \to & \text{rcv} & \to & (\mathcal{S}_B \times \mathcal{K}) \cup \{(\bot, \bot)\} \end{array}$$

Please note that de-randomizing (or explicitly considering the randomness of) the initialization of URKE is of little value since an adversary, when controlling the random coins of init, obtains all information necessary to compute all keys between A and B.

Correctness Intuitively a URKE scheme is correct, if all keys produced with send operations of A can also be obtained with the resulting ciphertext by the respective receive operations of B. More formally: Let $\{ad_i \in \mathcal{AD}\}_{i \geq 1}$ be a sequence of associated data. Let $\{st_{A,i}\}_{i \geq 0}, \{st_{B,i}\}_{i \geq 0}$ denote the sequences of A's and B's states generated by applying $\text{snd}(\cdot, ad_i)$ and $\text{rcv}(\cdot, ad_i, \cdot)$ operations iteratively for $i \geq 1$, that is, $(st_{A,i}, k_i, c_i) \leftarrow_{\$} \text{snd}(st_{A,i-1}, ad_i)$ and $(st_{B,i}, k'_i) \leftarrow \text{rcv}(st_{B,i-1}, ad_i, c_i)$. We say URKE scheme $\mathsf{UR} = (\text{init}, \text{snd}, \text{rcv})$ is correct if for all $st_{A,0}, st_{B,0} \leftarrow_{\$} \text{init}$, for all associated-data sequences $\{ad_i\}_{i \geq 1}$, and for all random coins used for snd calls, the key sequences $\{k_i\}_{i \geq 1}$ and $\{k'_i\}_{i \geq 1}$ generated as above are equal.

We note that we here explicitly require functionality (i.e., that B can recover *all* keys established by A under a passive adversary) in our correctness definition (in contrast to Figure 3.5).

Security For security, we provide the KINDR game for defining key indistinguishability under randomness manipulation of URKE in Figure 4.6. In this game, the adversary can let the session participants A and B send and receive ciphertexts via SndA and RcvB oracle queries, respectively, to establish keys between them. By querying the Reveal

or Challenge oracles, the adversary can obtain these established keys or receive a challenge key (that is, either the real established key or a randomly sampled element from the key space), respectively. Finally, the adversary can expose A's and B's state as the output of a specified send or receive operation, respectively, via oracles ExposeA or ExposeB.

When querying the SndA oracle, the adversary can specify the random coins rc for the invocation of the snd algorithm from the set $\mathcal{R}$ or indicate that it wants the random coins to be sampled uniformly at random by letting $rc = \epsilon$. By allowing the adversary to set the randomness for the invocations of the snd algorithm and exposing past states (which was not permitted in game KIND from Figure 3.6), new trivial attacks arise.

Below we review and explain the trivial attacks of the original URKE KIND game from Section 3.3, map them to the KINDR game here, and then introduce new trivial attacks that arise due to randomness manipulation.

A conceptual difference between our game definition here and the games from Chapter 3 is the way variables (especially arrays) are indexed. While the KIND games in figures 3.6, 3.9, and 3.12 make use of counters (of send and receive operations) to index computed keys and adversarial events, we here use the communicated transcripts, sent and received by A and B respectively, as indices. We thereby heavily exploit the fact that synchronicity (and divergence) of the communication between A and B are defined over these transcripts, which results for the stronger (and thereby more complex) adversary in a more comprehensible (but equivalent) game notation. Please note that, due to our indexing scheme, it suffices for our game definition to maintain a common key array K[·] and common sets of known keys KN and challenged keys CH for A and B (as opposed to arrays and sets for each party).[8]

[8] This is because a key, computed during the sending of A and the corresponding receiving of B, only differs between A and B, according to correctness, if the received transcript of B diverged from the sent transcript of A.

Game $\text{KINDR}^b_{\mathsf{UR}}(\mathcal{A})$
00 $\text{XP}_A \leftarrow \emptyset$; $\text{MR} \leftarrow \emptyset$
01 $\text{KN} \leftarrow \emptyset$; $\text{CH} \leftarrow \emptyset$
02 $trs \leftarrow \epsilon$; $trr \leftarrow \epsilon$
03 $\text{ST}_A[\cdot] \leftarrow \bot$; $\text{ST}_B[\cdot] \leftarrow \bot$
04 $\text{K}[\cdot] \leftarrow \bot$;
05 $(st_A, st_B) \leftarrow_{\$} \text{init}$
06 $\text{ST}_A[trs] \leftarrow st_A$; $\text{ST}_B[trr] \leftarrow st_B$
07 $b' \leftarrow_{\$} \mathcal{A}$
08 · Require $\text{KN} \cap \text{CH} = \emptyset$
09 Stop with b'

Oracle $\text{SndA}(ad, rc)$
10 Require $ad \in \mathcal{AD} \wedge rc \in \mathcal{R} \cup \{\epsilon\}$
11 If $rc = \epsilon$:
12 $(st_A, k, c) \leftarrow_{\$} \text{snd}(st_A, ad)$
13 Else:
14 $(st_A, k, c) \leftarrow \text{snd}(st_A, ad; rc)$
15 ○ $\text{MR} \xleftarrow{\cup} \{trs \| (ad, c)\}$
16 ○ If $trs \in \text{XP}_A$:
17 ○ $\text{KN} \xleftarrow{\cup} \{trs \| (ad, c)\}$
18 ○ $\text{XP}_A \xleftarrow{\cup} \{trs \| (ad, c)\}$
19 $trs \xleftarrow{\shortparallel} (ad, c)$
20 $\text{K}[trs] \leftarrow k$; $\text{ST}_A[trs] \leftarrow st_A$
21 Return c

Oracle $\text{Reveal}(tr)$
22 Require $\text{K}[tr] \in \mathcal{K}$
23 · $\text{KN} \xleftarrow{\cup} \{tr\}$
24 Return $\text{K}[tr]$

Oracle $\text{RcvB}(ad, c)$
25 Require $ad \in \mathcal{AD} \wedge c \in \mathcal{C} \wedge st_B \neq \bot$
26 · If $trr \| (ad, c) \not\preceq trs$
$\wedge \text{LCP}(trs, trr) \in \text{XP}_A$:
27 · $\text{KN} \xleftarrow{\cup} \{trr \| (ad, c)\}$
28 $(st_B, k) \leftarrow \text{rcv}(st_B, ad, c)$
29 If $k = \bot$: Return $\bot$
30 $trr \xleftarrow{\shortparallel} (ad, c)$
31 $\text{K}[trr] \leftarrow k$; $\text{ST}_B[trr] \leftarrow st_B$
32 Return

Oracle $\text{ExposeA}(tr)$
33 Require $\text{ST}_A[tr] \in \mathcal{S}_A$
34 · $\text{XP}_A \xleftarrow{\cup} \{tr\}$
35 ○ $trace \leftarrow \{tr^* \in \mathcal{TR}^* : \forall tr' \in \mathcal{TR}^*$
$(tr \prec tr' \preceq tr^* \implies tr' \in \text{MR})\}$
36 ○ $\text{KN} \xleftarrow{\cup} trace$; $\text{XP}_A \xleftarrow{\cup} trace$
37 Return $\text{ST}_A[tr]$

Oracle $\text{ExposeB}(tr)$
38 Require $\text{ST}_B[tr] \in \mathcal{S}_B$
39 · $\text{KN} \xleftarrow{\cup} \{tr^* \in \mathcal{TR}^* : tr \prec tr^*\}$
40 Return $\text{ST}_B[tr]$

Oracle $\text{Challenge}(tr)$
41 Require $\text{K}[tr] \in \mathcal{K}$
42 · Require $tr \notin \text{CH}$
43 $k \leftarrow b \,?\, \text{K}[tr] : \$(\mathcal{K})$
44 · $\text{CH} \xleftarrow{\cup} \{tr\}$
45 Return k

Figure 4.6: Games KINDR^b, $b \in \{0, 1\}$, for URKE scheme UR. Lines of code tagged with a '·' denote mechanisms to prevent or detect trivial attacks without randomness manipulation; trivial attacks caused by randomness manipulation are detected and prevented by lines tagged with '○'. We define $\text{LCP}(X, Y)$ to return the longest common prefix between X and Y, which are lists of atomic elements $z_i \in (\mathcal{AD} \times \mathcal{C})$. By longest common prefix we mean the longest list $Z = z_0 \| \ldots \| z_n$ for which $Z \preceq X \wedge Z \preceq Y$. We further define $\mathcal{TR} = \mathcal{AD} \times \mathcal{C}$. Line 39 is a shortcut notion that can be implemented efficiently. XP: exposed states, MR: states and keys affected by manipulated randomness, KN: known keys, CH: challenge keys, trs, trr: transcripts.

The lines marked with '·' in Figure 4.6 denote the handling of trivial attacks without randomness manipulation (as in Figure 3.6). Lines marked with '∘' introduce modifications that become necessary due the new trivial attacks based on manipulation of randomness.

Trivial attacks without randomness manipulations are:

(a) If the adversary reveals a key via oracle Reveal, then challenging this key via oracle Challenge is trivial. In order to prevent reveal and challenge of the same key, sets KN and CH trace which keys have been revealed (line 23) and challenged (line 44). The adversary only wins, if the intersection of both sets is empty (line 08). Additionally, a key must only be challenged once as otherwise bit b can be obtained trivially (line 42). Example: $c \leftarrow \mathrm{SndA}(\epsilon, \epsilon)$; $k \leftarrow \mathrm{Reveal}((\epsilon, c))$; output $[k = \mathrm{Challenge}((\epsilon, c))]$

(b) As keys, that are computed by both parties (because ciphertexts between them have not been manipulated yet), are stored only once in array K (due to the indexing of arrays with transcripts instead of pure counters), the adversary cannot reveal these keys on one side of the communication (e.g., at A) and then challenge them on the other side (e.g., at B). Consequently, this trivial attack (which was explicitly considered in Section 3.3) is implicitly handled by our game definition.

(c) After exposing B's state via oracle ExposeB, the adversary can comprehend all future computations of B. Consequently, all keys that can be received by B in the future are marked known (line 39). Example: $st_B \leftarrow \mathrm{ExposeB}(\epsilon)$; $c \leftarrow \mathrm{SndA}(\epsilon, \epsilon)$; $\mathrm{RcvB}(\epsilon, c)$; $(st_B, k) \leftarrow \mathrm{rcv}(st_B, \epsilon, c)$; output $[k = \mathrm{Challenge}((\epsilon, c))]$

(d) Exposing B's state, as long as the communication between A and B has not yet been manipulated by the adversary, allows the adversary also to compute all future keys established by A (which is also implicitly handled by our indexing of arrays via transcripts).

(e) Exposing A's state via oracle ExposeA allows the adversary to impersonate A towards B by using the exposed state to create and send own valid ciphertexts to B. As creating a forged ciphertext reveals the key that is computed by B on receipt, such keys are marked known (lines 26-27). The detection of this trivial attack works as follows: As soon as B receives a ciphertext that was not sent by A (i.e., B's transcript together with the received ciphertext is not a prefix of A's transcript) and A was exposed after A sent the last ciphertext that was also received by B (i.e., after the last common prefix LCP), the adversary is able to create this ciphertext validly on its own.[9] Example: $st_A \leftarrow \text{ExposeA}$; $(st_A, k, c) \leftarrow \text{snd}(st_A, \epsilon)$; $\text{RcvB}(\epsilon, c)$; output $[k = \text{Challenge}((\epsilon, c))]$

Due to randomness manipulations, the adversary can additionally conduct the following attacks trivially:

(f) If the randomness for sending is set by the adversary (via $\text{SndA}(ad, rc), rc \neq \epsilon$) and the state, used for this sending, is exposed (via ExposeA), then also the next state of A, output by this send operation, will be known (and marked as exposed) as sending is thereby deterministically computed on inputs that are known by the adversary (lines 16,18). Since the adversary can also retrospectively expose A's state, all computations that can be traced, due to continuous manipulated randomness of subsequent SndA oracle queries (unified in set MR) after such an exposure, are also marked as exposed (lines 35-36). Example: $rc \leftarrow_{\$} \mathcal{R}$; $c' \leftarrow \text{SndA}(\epsilon, rc)$; $\text{RcvB}(\epsilon, c')$; $st_A \leftarrow \text{ExposeA}(\epsilon)$; $(st_A, k', c') \leftarrow \text{snd}(st_A, \epsilon; rc)$; $(st_A, k, c) \leftarrow_{\$} \text{snd}(st_A, \epsilon)$; $\text{RcvB}(\epsilon, c)$; output $[k = \text{Challenge}((\epsilon, c')\|(\epsilon, c))]$

[9] Please note that we need to detect this trivial attack this way (in contrast to the game from Figure 3.6) because the adversary can forge ciphertexts to B without letting the communication between A and B actually diverge. It can do so by creating an own valid ciphertext which it sends to B (via $st_A \leftarrow \text{ExposeA}(\epsilon)$; $rc \leftarrow_{\$} \mathcal{R}$; $(st_A, k, c) \leftarrow \text{snd}(st_A, \epsilon; rc)$; $\text{RcvB}(\epsilon, c)$) but then it lets A compute the same ciphertext (via $\text{SndA}(\epsilon, rc)$). As a result, A and B are still in sync.

(g) Similarly, if the randomness for sending is set by the adversary and the state that A uses during this send operation is exposed, then the key, computed during sending, is known by the adversary since its computation is thereby deterministic (lines 16-17,35-36). Example: $rc \leftarrow_{\$} \mathcal{R}$; $c \leftarrow \mathrm{SndA}(\epsilon, rc)$; $st_A \leftarrow \mathrm{ExposeA}(\epsilon)$; $(st_A, k, c) \leftarrow \mathrm{snd}(st_A, \epsilon; rc)$; output $[k = \mathrm{Challenge}((\epsilon, c))]$

Based on this game, we define the advantage of an adversary in breaking the security of an URKE scheme as follows: The advantage of an adversary $\mathcal{A}$ against a URKE scheme UR in game KINDR from Figure 4.6 is $\mathrm{Adv}^{\mathrm{kindr}}_{\mathsf{UR}}(\mathcal{A}) = \left|\Pr[\mathrm{KINDR}^0_{\mathsf{UR}}(\mathcal{A}) = 1] - \Pr[\mathrm{KINDR}^1_{\mathsf{UR}}(\mathcal{A}) = 1]\right|$. We say that an URKE scheme UR is secure if the advantage is negligible for all probabilistic polynomial time adversaries $\mathcal{A}$.

Please note that KINDR security of URKE is strictly stronger than both KIND security notions of URKE, defined by Bellare et al. [BSJ+17] and presented in Section 3.3 (which themselves are incomparable among each other).

4.4 kuKEM* to URKE

Since our ultimate goal is to show that existence of a kuKEM* primitive is a necessary and sufficient condition to construct a URKE primitive—albeit requiring the help of hash functions (modeled as random oracle)—, we dedicate this section to indicate how to prove the latter of these implications.

Construction of URKE from kuKEM* We give a generic way to construct a URKE scheme UR from a kuKEM* scheme K with the help of random oracle H and MAC scheme M. This transformation $\mathsf{K} \to \mathsf{UR}$ is fully depicted in Figure 4.7. Below we briefly describe the algorithms of URKE scheme $\mathsf{UR} = (\mathrm{init}, \mathrm{snd}, \mathrm{rcv})$.

During the state initiation algorithm init, a kuKEM* key pair (sk, pk) is generated such that the encapsulation key pk is embedded into the sender state st_A, and the decapsulation key sk into the receiver state

st_B. The remaining state variables are exactly same for A and B. More specifically, two further keys are generated during initialization: the symmetric chaining key $k.c$ and a MAC key $k.m$. Furthermore the sent or received transcript (initialized with an empty string ϵ) is stored in each state. For brevity, we define that $k.c$, $k.m$, and the update key $k.u$ (used during sending and receiving; see below) all belong to the same key domain $\mathcal{K}$.

Proc init	**Proc** snd(st_A, ad)	**Proc** rcv(st_B, ad, C)
00 · $(sk, pk) \leftarrow_{\$} \mathrm{gen}_{\mathsf{K}}$	06 $(pk, k.c, k.m, t) \leftarrow st_A$	15 $(sk, k.c, k.m, t) \leftarrow st_B$
01 $k.c \leftarrow_{\$} \mathcal{K}$; $k.m \leftarrow_{\$} \mathcal{K}$	07 · $(pk, k, c) \leftarrow_{\$} \mathrm{enc}(pk)$	16 $c \,\|\|\, \tau \leftarrow C$
02 $t \leftarrow \epsilon$	08 $\tau \leftarrow \mathrm{tag}(k.m, ad \,\|\|\, c)$	17 Require $\mathrm{vfy}_{\mathsf{M}}(k.m, ad \,\|\|\, c, \tau)$
03 $st_A \leftarrow (pk, k.c, k.m, t)$	09 $C \leftarrow c \,\|\|\, \tau$	18 · $(sk, k) \leftarrow \mathrm{dec}(sk, c)$
04 $st_B \leftarrow (sk, k.c, k.m, t)$	10 $t \xleftarrow{\shortparallel} ad \,\|\|\, C$	19 Require $k \neq \perp$
05 Return (st_A, st_B)	11 · $k.o \,\|\|\, k.c \,\|\|\, k.m \,\|\|\, k.u \leftarrow \mathrm{H}(k.c, k, t)$	20 $t \xleftarrow{\shortparallel} ad \,\|\|\, C$
	12 · $pk \leftarrow \mathrm{up}(pk, k.u)$	21 · $k.o \,\|\|\, k.c \,\|\|\, k.m \,\|\|\, k.u \leftarrow \mathrm{H}(k.c, k, t)$
	13 $st_A \leftarrow (pk, k.c, k.m, t)$	22 · $sk \leftarrow \mathrm{up}(sk, k.u)$
	14 Return $(st_A, k.o, C)$	23 $st_B \leftarrow (sk, k.c, k.m, t)$
		24 Return $(st_B, k.o)$

Figure 4.7: Construction of a URKE scheme from a kuKEM* scheme $\mathsf{K} = (\mathrm{gen}_{\mathsf{K}}, \mathrm{up}, \mathrm{enc}, \mathrm{dec})$, a message authentication code $\mathsf{M} = (\mathrm{tag}, \mathrm{vfy}_{\mathsf{M}})$, and a random oracle H. For simplicity we denote the key space of the MAC and the space of the chaining key $k.c$ in st_A with the same symbol $\mathcal{K}$. Lines marked with a '·' differ from our URKE scheme in Figure 3.7.

On sending, public key pk in A's state is used by the encapsulation algorithm to generate key k and ciphertext c. Then, MAC key $k.m$, contained in the current state of A, is used to issue a tag τ over the tuple of associated data ad and encapsulation ciphertext c. The finally sent ciphertext, denoted by C, is a concatenation of c and the MAC tag τ. The output key $k.o$, as well as the symmetric keys of the next state of A are obtained from the random oracle, on input of the chaining key $k.c$, the freshly encapsulated key k, and the history of sent transcript t. Finally, a kuKEM* update is applied on pk under associated data that is derived from the random oracle output (denoted by $k.u$). Please note that the encapsulation algorithm is the only randomized operation inside snd. Hence the random coins of the

latter are only used by this encapsulation.

On receiving, the operations are on par with the sending algorithm. Namely, the received ciphertext C is parsed as the encapsulation ciphertext c and the MAC tag τ. The latter is verified with regards to the MAC key $k.m$, stored in the state of B. After the key k is decapsulated, the same input to the random oracle H is composed. The symmetric components of the next state and $k.o$ are derived from the random oracle's output. Finally, the secret key sk is updated with $k.u$, so that it is in-sync with the update of pk.

We remark that our construction in Figure 4.7 differs from the one in Figure 3.7 only in the output of the random oracle and in the subsequent use of the kuKEM*'s update algorithm (instead, the latter freshly generates a new KEM key pair from the random oracle output). These changes are, nevertheless, significant as the scheme from Section 3.4 is insecure when the adversary is able to (reveal or) manipulate the random coins for invocations of the snd algorithm. We give an according detailed attack description with respect to our KINDR model, introduced here, in Section 4.4.1.

Theorem 4 *The URKE protocol* UR *from Figure 4.7 offers key indistinguishability under randomness manipulation. More precisely, if function* H *is modeled as a random oracle, for every adversary* $\mathcal{A}$ *against URKE scheme* UR *in games* $\mathrm{KINDR}^b_{\mathsf{UR}}$ *from Figure 4.6 there exists an adversary* $\mathcal{B}_{\mathsf{K}}$ *against* kuKEM* *scheme* K *in game* KUOWR *from Figure 4.4 and an adversary* $\mathcal{B}_{\mathsf{M}}$ *against MAC* M *in game* SUF *from Figure 2.2 such that*

$$\mathrm{Adv}^{\mathrm{kindr}}_{\mathsf{UR}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathrm{kuowr}}_{\mathsf{K}}(\mathcal{B}_{\mathsf{K}}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}}) + \frac{q_{\mathrm{H}} \cdot (q_{\mathrm{SndA}} + q_{\mathrm{RcvB}})}{|\mathcal{K}|}$$

, where $\mathcal{K}$ *is the key domain in the construction* UR*,* q_{SndA}*,* q_{RcvB}*, and* q_{H} *are the number of* SndA*,* RcvB *and* H *queries respectively by* $\mathcal{A}$*, and the running time of* $\mathcal{A}$ *is approximately the running time of* $\mathcal{B}_{\mathsf{K}}$ *and* $\mathcal{B}_{\mathsf{M}}$*.*

The proof of Theorem 4 is a slight adaption of the one from Section 3.10. This adaption has not been contributed by the author of

this thesis. Hence, we refer the interested reader to Section 3.10 or to the article on which this chapter bases [BRV20a, BRV20b].

4.4.1 Weaknesses of Previous URKE Schemes

As described before, both previous unidirectional RKE security definitions are slightly weaker than ours, allowing the instantiations to bypass our equivalence result. In the following we describe (non-trivial) attacks according to our security definition against both schemes.

Due to only allowing exposures of A in the unidirectional RKE security definition of Bellare et al. [BSJ+17], no forward secure state update for B is required during invocations of the rcv algorithm. Accordingly, when exposing the state of B (which contains a static secret key) in the scheme by Bellare et al., all established keys between A and B become insecure (as opposed to only keys established with this exposed or future states). Example: $c \leftarrow \text{SndA}(\epsilon, \epsilon)$; $\text{RcvB}(\epsilon, c)$; $c' \leftarrow \text{SndA}(\epsilon, \epsilon)$; $\text{RcvB}(\epsilon, c')$; $st_B \leftarrow \text{ExposeB}((\epsilon, c) \| (\epsilon, c'))$; $(X, \sigma) \leftarrow c$; $(hk, y, i, \dots) \leftarrow st_B$; $k \leftarrow \text{H}(hk, (i, \sigma, X, X^y))$; output $[k = \text{Challenge}((\epsilon, c))]$

While allowing the adversary to expose B's state in our unidirectional RKE security definition from Section 3.3, we there do not consider randomness reveal (nor manipulation of randomness). Our according unidirectional RKE scheme exploits this in the state update of A and B during the invocation of snd and rcv respectively. The new state of A during the snd invocation is derived by generating a new KEM key pair based on the randomness of this invocation together with the secrets in the previous state of A. The secret key of this KEM key pair is immediately discarded and only the public key is stored in A's state. B derives the corresponding secret key from the ciphertext that he receives from A and his previous state secrets. As a consequence, an adversary obtains B's current secret key (and thereby all his future secret keys) as soon as it once knows the current state of A and then manipulates randomness for the subsequent invocation of snd. Example: $st_A \leftarrow \text{ExposeA}(\epsilon)$; $(rc_0 \| rc_1) \leftarrow_{\$} \mathcal{R}$; $c' \leftarrow \text{SndA}(\epsilon, rc_0 \| rc_1)$; $c \leftarrow \text{SndA}(\epsilon, \epsilon)$; $(pk, k.c, k_m, t) \leftarrow st_A$; $(k'_e, c'_e) \leftarrow \text{enc}(pk; rc_0)$; $t \leftarrow (\epsilon, c')$; $(k', k.c, k_m, sk) \leftarrow \text{H}(k.c, k_e, t)$; $(c_e, \tau) \leftarrow c$; $k_e \leftarrow \text{dec}(sk, c_e)$; $t \xleftarrow{\shortparallel} (\epsilon, c)$;

$(k, k.c, k_m, sk) \leftarrow \mathrm{H}(k.c, k_e, t)$; output $[k = \mathrm{Challenge}((\epsilon, c')\|(\epsilon, c))]$

4.4.2 Insufficiency of Weakly Updatable PKE

Apart from our kuKEM notion from Section 3.2 (which is similar to kuPKE by Jaeger and Stepanovs [JS18a]) and our enhanced kuKEM* notion from Section 4.2, Jost et al. [JMM19] introduced the notion of healable and key-updating public-key encryption (HkuPke). The former three are instantiated from HIBE and the latter can be derived from efficient building-blocks based on Diffie–Hellman assumptions.

Intuitively, the key update mechanism in kuKEM* (and kuKEM) depicts a one-way function that can be applied independently on secret key and public key with respect to some associated data. It is yet unclear, how to implement this mechanism without relying on HIBE primitives.

In contrast, the intuition behind the update mechanism in HkuPke depicts a merging of an old key pair with some update key pair (implemented via multiplying public Diffie–Hellman shares and adding their secret exponents respectively), and a deterministic deriving of new key pairs. Problems with this mechanism are that the merging is not one-way (i.e., it can be inverted) and it either requires interaction from secret key holder to public key holder (for transmitting the update public key), or the public key holder learns the secret update exponent during the update.[10]

Why One-Wayness is Needed in URKE It is essential for KINDR security of URKE that the following attack is harmless with respect to the security of key k for all random coins $rc, rc^* \in \mathcal{R}, rc \neq rc^*$: $st_A^0 \leftarrow \mathrm{ExposeA}(\epsilon)$; $ad \leftarrow \epsilon$; $c^1 \leftarrow \mathrm{SndA}(ad, rc)$; $(st_A^1, k^1, c^1) \leftarrow \mathrm{snd}(st_A^0, ad; rc)$; $(st_A^*, k^*, c^*) \leftarrow \mathrm{snd}(st_A^0, ad; rc^*)$; $\mathrm{RcvB}(ad, c^*)$; $st_B^* \leftarrow \mathrm{ExposeB}((ad, c^*))$; $c^2 \leftarrow \mathrm{SndA}(ad, \epsilon)$; $st_A^2 \leftarrow \mathrm{ExposeA}((ad, c^1)\|(ad, c^2))$; $k \leftarrow \mathrm{Challenge}((ad, c^1)\|(ad, c^2))$

[10] When taking a look at the details in [JMM19]: The former is the case for their first public key ek^{upd} which is transmitted, and the latter is the case for their second public key ek^{eph} which is derived together with its secret key on the sender side.

The state updates due to oracle queries SndA(ad, rc) and RcvB(ad, c^*) (i.e., in the respective algorithm invocations of snd and rcv) must diverge states st_A^1 and st_B^* such that they cannot be used by an adversary to derive the key encapsulated in c^2. Note that the random coins rc^* can be arbitrarily chosen (e.g., similar to rc) in order to let c^* differ from c^1 only in few bits. Since any difference between c^1 and c^* must result in a divergence of st_B^* from a state that can be used to derive the key encapsulated in c^2, the secrets in B's state must be updated on the entire incoming ciphertext. This requires a 'one-way' update of the secrets in B's state that takes c^* as respected (associated) data. Furthermore, all computations for deriving st_A^1 and st_A^* can be traced and determined by the adversary, since rc and rc^* are chosen by it. Hence, the public key update (in this state derivation) is computed publicly (and cannot rely on any inputs from B), and the secret key update (in the derivation of st_B^*) must respect all incoming ciphertext bits.

Consequently, the public key update is deterministic (or probabilistic on adversarially chosen random coins) and based on public inputs, and a secret key update is based on adversarially chosen (ciphertext as) associated data, which is what the notion of KUOWR secure kuKEM* (and KUOW secure kuKEM) reflects but the syntax of HkuPku does not allow.

We finally remark: Jost et al. [JMM19] explicitly make it transparent that they do not aim to protect against such attacks as their main goal is an efficient protocol but not optimal security. Hence, we do not label it a weakness of their primitive but only an important and notable difference.

4.5 URKE to kuKEM*

In order to show that public key encryption with independently updatable key pairs (in our case kuKEM*) is a necessary building block for ratcheted key exchange, we build the former from the latter. The major obstacle is that the updates of public key and secret key of a

kuKEM* are conducted independently—consequently no communication between holder of the public key and holder of the secret key can be exploited for updates. In contrast, all actions in ratcheted key exchange are based on communication (i.e., sent or received ciphertexts). Another property that public key updates for kuKEM* must fulfill—in contrast to state updates in KIND secure unidirectional RKE as in Section 3.3—is that they must not leak any information on the according secret key during the update computation. In the following we first explain these sketched issues (and their origin) in more detail, then describe how we solve it, and present a reduction of KUOWR security to KINDR security of a generic URKE scheme.

Crucial Properties of kuKEM* Syntax and KUOWR security of kuKEM* (as well as KUOW security of kuKEM) have several implications that we explain below. As described before, the syntax of kuKEM* does not allow interactions between secret key holder and public key holder(s) to communicate information for the key parts' updates (see Figure 4.1). This condition originates from the utilization of kuKEM as a building block for the instantiation of sesquidirectional ratcheted key exchange (SRKE; see Section 3.5). This extended RKE notion requires the two communication participants' states to immediately become incompatible as soon as one of the participants receives a ciphertext that was manipulated by the adversary. Public key and secret key of the used kuKEM, as part of the respective state, are therefore updated independently in order to cause an immediate divergence between these key pair parts. A full description of the attack that is prevented by independent key updates can be found in Section 3.7.2.

A second property that immediately follows from the first one is that, for all public keys that are updated equally, a compatible secret key can be used to decapsulate ciphertexts from all these public keys. As a public key update can also be conducted by an adversary, the computation of this update itself must not reveal any information on encapsulated keys—especially not on a compatible secret key. We will

further comment on this property when explaining that, even though KINDR security of URKE implies KUOWR security of kuKEM*, one can instantiate URKE KIND securely with standard key encapsulation mechanisms. In Section 4.4.1, we describe why these prior instantiations of KIND secure URKE are insecure according to KINDR security.

When deriving the notion of kuKEM* and its KUOWR security, we take these properties into account, as the goal of this chapter is not to find the minimal building block for unidirectional RKE, but for RKE in general (e.g., also for sesquidirectional RKE).

Construction of kuKEM* from URKE The weaker KIND security of URKE from Section 3.3 already allows that the sender's state st_A can always be exposed without affecting the security of any established keys (as long as this exposed state is not used to impersonate A towards B). Consequently, A's pure state reveals no information on encapsulated keys nor on B's secret key(s). KIND security of URKE further implies that B's state only reveals information on keys that have not yet been computed by B (while earlier computed keys stay secure). One can imagine A's state consequently as the public part of a (stateful) key pair and B's state as the secret counterpart.

The two above mentioned crucial properties of KUOW(R) security are, however, not implied by KIND security when using st_A as the public key and st_B as the secret key of a kuKEM. Firstly, updating st_B (as part of receiving a ciphertext) requires that the ciphertext, generated during sending of A (and updating of st_A), is known by B but the syntax of kuKEM does not allow an interaction between public key holder and secret key holder. This issue can be solved by de-randomizing the snd algorithm. If A's state as part of the public key is updated via a de-randomized invocation of snd, the secret key holder can also obtain the ciphertext that A would produce for the same update (by invoking the de-randomized/deterministic snd) and then update st_B with this ciphertext via rcv. A conceptional depiction of this is in Figure 4.8. Thereby the secret key is defined to

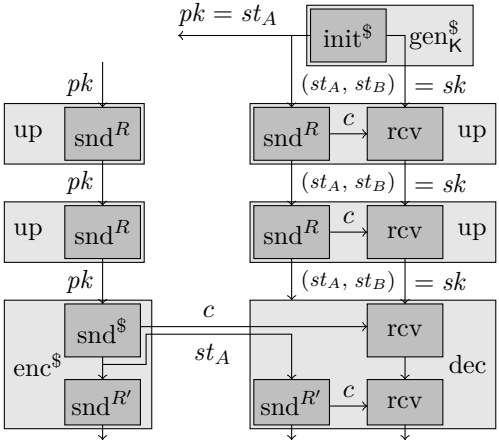

Figure 4.8: Conceptual depiction of kuKEM* construction from generic URKE scheme. The symbol in the upper index of an algorithm name denotes the source of random coins ('$' indicates uniformly sampled). R is a fixed value. For clarity we omit ad inputs and k outputs (cf. Figure 4.1).

contain st_A in addition to st_B.

Secondly, in the URKE construction from Section 3.4 A also temporarily computes secrets of B that match A's updated values during sending. As a result, normal KIND security allows that a derandomized snd invocation reveals the secrets of B to an adversary if st_A is known (see Section 4.4.1 for a detailed description of this attack). In order to solve this issue, the security definition of URKE must ensure that future encapsulated keys' security is not compromised if snd is invoked under a known state st_A and with random coins that are chosen by an adversary (i.e., KINDR security).

Our generic construction of a KUOWR secure kuKEM* from a generic KINDR secure URKE scheme is depicted in Figure 4.9. As described before, the public key contains state st_A and the secret key contains both states (st_A, st_B) that are derived from the init algorithm. In order to update the public key, the snd algorithm is invoked on state st_A, with the update associated-data, and fixed randomness. Output key and ciphertext are thereby ignored. Accordingly, the se-

cret key is updated by first invoking the snd algorithm on state st_A with the same fixed randomness and the update associated-data. This time the respective ciphertext from A to B is not omitted but used as input to rcv algorithm with the same associated data under st_B.

Encapsulation and decapsulation are conducted by invoking snd probabilistically and rcv respectively. In order to separate updates from en-/decapsulation, a '0' or '1' is prepended to the associated-data input of snd and rcv respectively. For bounding the probability of a ciphertext collision in the proof, a randomly sampled 'collision key' ck is attached to the associated data of the snd invocation in encapsulation. In order to accordingly add ck to the associated data of rcv as part of the decapsulation, ck is appended to the ciphertext. Since state st_A, output by the snd algorithm during the encapsulation, is computed probabilistically, it is also attached to the encapsulation ciphertext, so that (the other) st_A, embedded in the secret key, can be kept compatible with the public key holder's. To bind ck and st_A to the ciphertext, both are integrity protected by a message authentication code (MAC) that takes one part of the key from the snd invocation as MAC key (only the remaining key bytes are output as the encapsulated kuKEM* key). Additionally the whole ciphertext (i.e., URKE ciphertext, collision key, state st_A, and MAC tag) is used as associated data for an additional 'internal update' of public key and secret key in encapsulation and decapsulation, respectively. This is done to escalate manipulations of collision key, state st_A, or MAC tag (as part of the ciphertext) back into the URKE states st_A and st_B (as part of public key and secret key). For full details on the rationales behind these two *binding* steps we refer the reader to the proof.

Interestingly, the public key holder can postpone the de-randomized snd invocation for public key updates until encapsulation and instead only remember the updates' associated data without compromising security. However, the updates of the secret key must be performed immediately as otherwise an exposure of the current secret key reveals also information on its past versions. Thereby the computation of snd in algorithm up must be conducted during the secret key update without interaction between public key holder and secret key holder.

Proc gen_{K}
00 $(st_A, st_B) \leftarrow_{\$} \text{init}$
01 $pk \leftarrow st_A$
02 $sk \leftarrow (st_A, st_B)$
03 Return (pk, sk)

Proc $\text{up}(pk, ad)$
04 $(pk, \_, \_) \leftarrow \text{snd}(pk, (0, ad); 0)$
05 Return pk

Proc $\text{enc}(pk)$
06 $ck \leftarrow_{\$} \mathcal{K}$
07 $(pk, (k, k.m), c') \leftarrow_{\$} \text{snd}(pk, (1, ck))$
08 $\tau \leftarrow \text{tag}(k.m, (ck, pk, c'))$
09 $c \leftarrow (ck, pk, c', \tau)$
10 $(pk, \_, \_) \leftarrow \text{snd}(pk, (2, c); 0)$
11 Return (pk, k, c)

Proc $\text{up}(sk, ad)$
12 $(st_A, st_B) \leftarrow sk$
13 $(st_A, \_, c) \leftarrow \text{snd}(st_A, (0, ad); 0)$
14 $(st_B, \_) \leftarrow \text{rcv}(st_B, (0, ad), c)$
15 $sk \leftarrow (st_A, st_B)$
16 Return sk

Proc $\text{dec}(sk, c)$
17 $(st_A, st_B) \leftarrow sk$
18 $(ck, pk, c', \tau) \leftarrow c$
19 $(st_B, (k, k.m)) \leftarrow \text{rcv}(st_B, (1, ck), c')$
20 Require $\text{vfy}_{\mathsf{M}}(k.m, (ck, pk, c'), \tau)$
21 $(st_A, \_, c'') \leftarrow \text{snd}(pk, (2, c); 0)$
22 $(st_B, \_) \leftarrow \text{rcv}(st_B, (2, c), c'')$
23 $sk \leftarrow (st_A, st_B)$
24 Return (sk, k)

Figure 4.9: Construction of a key-updatable KEM from a generic URKE scheme $\mathsf{UR} = (\text{init}, \text{snd}, \text{rcv})$ and one-time message authentication code $\mathsf{M} = (\text{tag}, \text{vfy}_{\mathsf{M}})$.

Theorem 5 *If URKE scheme* UR *is* KINDR *secure according to Figure 4.6, one-time MAC* M *is* SUF *secure according to Figure 2.2, and for all* $(k, k.m) \in \mathcal{K}_{\mathsf{UR}}$ *it holds that* $k \in \mathcal{K}_{\mathsf{K}}$ *and* $k.m \in \mathcal{K}_{\mathsf{M}}$*, then* kuKEM* *scheme* K *from Figure 4.9 is* KUOWR *secure according to Figure 4.4 with*

$$\text{Adv}^{\text{kuowr}}_{\mathsf{K}}(\mathcal{A}) \leq q_{\text{Gen}} q_{\text{Enc}} \cdot \left(\text{Adv}^{\text{kindr}}_{\mathsf{UR}}(\mathcal{B}_{\mathsf{UR}}) + \text{Adv}^{\text{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}}) + \frac{1}{|\mathcal{K}|} \right),$$
$$\textit{with } \text{Adv}^{\text{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}}) \leq \text{Adv}^{\text{kindr}}_{\mathsf{UR}}(\mathcal{B}_{\mathsf{UR}})$$

where $\mathcal{A}$ *is an adversary against* KUOWR *security,* $\mathcal{B}_{\mathsf{UR}}$ *is an adversary against* KINDR *security of* UR*,* $\mathcal{B}_{\mathsf{M}}$ *is an adversary against* SUF *security of* M*,* q_{Gen} *and* q_{Enc} *are the number of* Gen *and* Enc *queries by* $\mathcal{A}$ *respectively,* $\mathcal{K}$ *is the space from which* ck *is sampled, and the running time of* $\mathcal{A}$ *is approximately the running time of* $\mathcal{B}_{\mathsf{UR}}$ *and* $\mathcal{B}_{\mathsf{M}}$*.*

In Section 4.5.2 we show how to construct an SUF secure one-time MAC from a generic KINDR secure URKE scheme, which implies

the second term in Theorem 5. We prove Theorem 5 below and provide a formal pseudo-code version of the simulation's game hops in Figure 4.10.

4.5.1 Proof of URKE to kuKEM*

We conduct the proof in four game hops: In the first game hop we guess for which instance the first valid Solve oracle query is provided by the adversary; in the second game hop, we guess for which Enc oracle query of the previously guessed instance the first valid Solve oracle query is provided; additionally the simulation aborts in this game hop if the adversary crafts this first valid ciphertext and provides it to the Dec oracle before it is output by the Enc oracle; in the third game hop, we replace the key, output by the first snd invocation in this guessed Enc oracle query by a randomly sampled key (which is reduced to KINDR security of UR); in the final game hop, we abort on a MAC forgery, provided to the Dec oracle, that belongs to the ciphertext that is output by the guessed Enc oracle query (which is reduced to SUF security of M).

Game 0 This game is equivalent to the original KUOWR game.

Game 1 The simulation guesses for which instance n_{Gen} the first key k^* is provided to the Solve oracle such that the secret key for decapsulation is not marked exposed (i.e., $tr^* \notin \text{XP}_{n_{\text{Gen}}}$) and the provided key equals the indicated challenge key (i.e., $k^* = \text{CK}_{n_{\text{Gen}}}[tr^*]$). Therefore n_{Gen} is randomly sampled from $[q_{\text{Gen}}]$, where q_{Gen} is the number of Gen oracle queries by the adversary. The reduction aborts if n_{Gen} is not the instance for which the first valid Solve oracle query is provided (see Figure 4.10 lines 48,51).

Consequently we have $\text{Adv}^{G_0} = q_{\text{Gen}} \cdot \text{Adv}^{G_1}$.

Game 2 The simulation guesses in which of n_{Gen}'s Enc queries the challenge is created, that is the first valid query to the Solve oracle by the adversary. Therefore n_{Enc} is randomly sampled from $[q_{\text{Enc}}]$ and the simulation aborts if either the randomness for the n_{Enc}'s Enc query is manipulated as thereby no challenge would be created (lines 28,29), or the first valid query to the Solve oracle is for another challenge than

the one created by n_{Gen}'s n_{Enc}th Enc query (lines 49,51), or a secret key that helps to trivially solve the challenge from n_{Gen}'s n_{Enc}th Enc query is exposed (lines 29,82).

In addition, the simulation aborts if, before the n_{Gen}'s n_{Enc}th Enc query was made, Dec was queried on a ciphertext (with the same preceding transcript) that contains the same URKE ciphertext and 'collision key' ck as n_{Gen}'s n_{Enc}th Enc query (lines 28,29). As the probability of a collision in the URKE transcript (i.e., associated data and ciphertext of the first snd invocation of n_{Gen}'s n_{Enc}th Enc query were previously already provided to n_{Gen}'s n_{Enc}th Dec query under the same preceding transcript) is bounded by a collision in the the key space $\mathcal{K}$ (as thereby ck as associated data must collide), we have $\text{Adv}^{G_1} = q_{\text{Enc}} \cdot \left(\text{Adv}^{G_2} + \frac{1}{|\mathcal{K}|}\right)$.

Game 3 The simulation replaces the output $(k, k.m)$ from the first snd invocation of n_{Gen}'s n_{Enc}th Enc query by values randomly sampled.

An adversary that can distinguish between **Game 2** and **Game 3** can be turned into an adversary that breaks KINDR security of URKE scheme UR. The reduction is as follows´: The reduction obtains n_{Gen}'s public key in oracle Gen via oracle ExposeA from the KINDR game. Invocations of snd in Up_S to n_{Gen} are replaced by SndA and ExposeA queries. Invocations of snd in Up_R to n_{Gen} are processed by the reduction itself and the subsequent rcv invocations are replaced by RcvB queries. The state st_B in queries to Expose for n_{Gen} is obtained via ExposeB queries to the KINDR game. For all queries to Enc of n_{Gen} the snd invocations are replaced by SndA and ExposeA queries. kuKEM* key and MAC key $(k, k.m)$ for n_{Gen}'s Enc oracle queries are obtained via Reveal—except for n_{Gen}'s n_{Enc}th Enc query, in which these two keys are obtained from the Challenge oracle in the KINDR game. Invocations of rcv in the Dec oracle for n_{Gen} are replaced by RcvB queries and Reveal queries (in case the respective key was not already computed in the Enc oracle). The snd invocation in oracle Dec is directly computed by the reduction.

In order to show that manipulations of transcripts in the KUOWR

game manipulate equivalently the transcripts in the KINDR game (such that the state st_A in the public key diverges from state st_B in the secret key iff the transcripts $trs_{n_{\text{Gen}}}$ and $trr_{n_{\text{Gen}}}$ diverge), we define the translation array $\mathrm{T}[\cdot]$ that maps the transcript of n_{Gen} in the KUOWR game to the according transcripts in the KINDR game.

As **Game 2** aborts if n_{Gen}'s n_{Enc}th Enc query entails no valid KINDR challenge, or if the respective ciphertext was already crafted by the adversary (and provided to the Dec oracle), an adversary, distinguishing the real key pair $(k, k.m)$ from the randomly sampled one, breaks KINDR security. Formally, the solution for n_{Gen}'s n_{Enc}th Enc query to the Solve oracle is compared with the challenge key k from the KINDR Challenge oracle (which is obtained during n_{Gen}'s n_{Enc}th Enc query): If the keys equal, the reduction terminates with $b' = 0$ (as thereby the KINDR game's challenge entailed the real key), otherwise it terminates with $b' = 1$.

Consequently we have $\mathrm{Adv}^{G_2} \leq \mathrm{Adv}^{G_3} + \mathrm{Adv}^{\text{kindr}}_{\mathsf{UR}}(\mathcal{B}_{\mathsf{UR}})$.

Game 4 The only way, the adversary can win in **Game 3**, is to keep secret key and public key of n_{Gen} compatible (by updating them equivalently and forwarding all Enc queries to the Dec oracle) and then forwarding only the URKE ciphertext c' of n_{Gen}'s n_{Enc}th Enc query to the Dec oracle while manipulating parts of the remaining challenge ciphertext. Thereby the Dec oracle outputs the correct challenge key such that the adversary trivially wins.[11]

We therefore define **Game 4** to let the simulation abort if a forgery of the MAC tag for the challenge ciphertext is provided to the Dec oracle. Distinguishing between **Game 3** and **Game 4** can hence be reduced to SUF security of one-time MAC M. The reduction is as follows: Instead of sampling $k.m$ randomly, the MAC tag for n_{Gen}'s n_{Enc}th Enc query is derived from the Tag oracle of the SUF game. Since an abort requires that the URKE challenge ciphertext c' is in-

[11] Please note that after this manipulation, states st_A and st_B in public key and secret key, respectively, diverge, but the key, output by the Dec oracle, still equals the challenge key. In case, URKE ciphertext c' from the challenge ciphertext is already provided manipulately to the Dec oracle, the challenge key is already independent from the key, computed in the Dec oracle.

deed received in oracle Dec (and also the transcripts prior to this ciphertext equal for $trs_{n_{\mathrm{Gen}}}$ and $trr_{n_{\mathrm{Gen}}}$), the URKE key (containing $k.m$) equals. As a consequence, a crafted ciphertext (pk, c', τ), provided to the Dec oracle, is a forgery τ for message (pk, c') in the SUF game.

Consequently we have $\mathrm{Adv}^{G_3} \leq \mathrm{Adv}^{G_4} + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}})$.

As the challenge key from n_{Gen}'s n_{Enc}th Enc query is randomly sampled and cannot be derived from any other oracle, the advantage in winning **Game 4** is $\mathrm{Adv}^{G_4} = 0$.

Summing up the advantages above, we have:

$$\begin{aligned}\mathrm{Adv}^{\mathrm{kuowr}}_{\mathsf{K}}(\mathcal{A}) &\leq q_{\mathrm{Gen}} q_{\mathrm{Enc}} \cdot \left(\mathrm{Adv}^{\mathrm{kindr}}_{\mathsf{UR}}(\mathcal{B}_{\mathsf{UR}}) + \mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}}) + \frac{1}{|\mathcal{K}|} \right) \\ &\leq q_{\mathrm{Gen}} q_{\mathrm{Enc}} \cdot \left(2 \cdot \mathrm{Adv}^{\mathrm{kindr}}_{\mathsf{UR}}(\mathcal{B}_{\mathsf{UR}}) + \frac{1}{|\mathcal{K}|} \right)\end{aligned}$$

where $\mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{B}_{\mathsf{M}}) \leq \mathrm{Adv}^{\mathrm{kindr}}_{\mathsf{UR}}(\mathcal{B}_{\mathsf{UR}})$ follows from a SUF secure one-time MAC construction from a generic KINDR secure URKE scheme UR (which is described in Section 4.5.2). ■

Simulation $\mathcal{S}_{\mathsf{K}}(\mathcal{A})$
00 $n \leftarrow 0$
01 $n_{\text{Gen}} \leftarrow_{\$} [q_{\text{Gen}}]$ $\mathbf{G}_{\geq 1}$
02 $n_{\text{Enc}} \leftarrow_{\$} [q_{\text{Enc}}]$; $e \leftarrow 0$ $\mathbf{G}_{\geq 2}$
03 $tr^{\circ} \leftarrow \epsilon$; $c^{\circ} \leftarrow \bot$ $\mathbf{G}_{\geq 2}$
04 $k^{\bullet} \leftarrow \bot$; $k.m^{\bullet} \leftarrow \bot$; $ck^{\bullet} \leftarrow \bot$; $c^{\bullet} \leftarrow \bot$ $\mathbf{G}_{\geq 3}$
05 Invoke $\mathcal{A}$
06 Stop with 0

Oracle Gen
07 $n \leftarrow n + 1$
08 If $n = n_{\text{Gen}}$: $\mathrm{T}[\cdot] \leftarrow \bot$ $\mathbf{G}_{=3}$
09 $(st_A, st_B) \leftarrow_{\$}$ init
10 $pk_n \leftarrow st_A$; $sk_n \leftarrow (st_A, st_B)$
11 $\mathrm{CK}_n[\cdot] \leftarrow \bot$; $\mathrm{XP}_n \leftarrow \emptyset$
12 $trs_n \leftarrow \epsilon$; $trr_n \leftarrow \epsilon$
13 $\mathrm{SK}_n[\cdot] \leftarrow \bot$; $\mathrm{SK}_n[trr_n] \leftarrow sk_n$
14 Return pk_n

Oracle $\mathrm{Up}_{\mathcal{S}}(i, ad)$
15 Require $1 \leq i \leq n \wedge ad \in \mathcal{AD}$
16 $(pk_i, k, c) \leftarrow \mathrm{snd}(pk_i, (0, ad); 0)$
17 If $i = n_{\text{Gen}}$: $\mathbf{G}_{=3}$
18 $\quad \mathrm{T}[trs_i \| ad] \leftarrow \mathrm{T}[trs_i] \| ((0, ad), c)$ $\mathbf{G}_{=3}$
19 $trs_i \xleftarrow{\shortparallel} ad$
20 Return pk_i

Oracle $\mathrm{Enc}(i, rc)$
21 Require $1 \leq i \leq n$
22 Require $rc \in \mathcal{R} \cup \{\epsilon\}$
23 If $rc = \epsilon$: $mr \leftarrow \mathtt{F}$; $rc \leftarrow_{\$} \mathcal{R}$
24 Else: $mr \leftarrow \mathtt{T}$
25 $ck \leftarrow_{\$} \mathcal{K}$
26 $(pk_i, (k, k.m), c') \leftarrow_{\$} \mathrm{snd}(pk_i, (1, ck); rc)$
27 If $i = n_{\text{Gen}} \wedge e = n_{\text{Enc}}$: $\mathbf{G}_{\geq 2}$
28 $\quad$ If $mr = \mathtt{T} \vee (\exists pk' \in \mathcal{PK}, \tau' \in \mathcal{T} :$
$\quad\quad trs_i \| (ck, pk', c', \tau') \in \mathrm{XP}_i$
$\quad\quad \vee trs_i \| (ck, pk', c', \tau') \preceq trr_i)$: $\mathbf{G}_{\geq 2}$
29 $\quad\quad$ Abort $\mathbf{G}_{\geq 2}$
30 $\quad (ck^{\bullet}, c^{\bullet}) \leftarrow (ck, c')$ $\mathbf{G}_{\geq 3}$
31 $\quad (k^{\bullet}, k.m^{\bullet}) \leftarrow_{\$} \mathcal{K}_{\mathsf{UR}}$ $\mathbf{G}_{\geq 3}$
32 $\quad (k, k.m) \leftarrow (k^{\bullet}, k.m^{\bullet})$ $\mathbf{G}_{\geq 3}$
33 $\tau \leftarrow \mathrm{tag}(k.m, (ck, pk_i, c'))$
34 $c \leftarrow (ck, pk_i, c', \tau)$
35 $(pk_i, c'', k'') \leftarrow \mathrm{snd}(pk_i, (2, c); 0)$
36 If $i = n_{\text{Gen}}$: $\mathbf{G}_{\geq 2}$
37 $\quad$ If $e = n_{\text{Enc}}$: $\mathbf{G}_{\geq 2}$
38 $\quad\quad tr^{\circ} \leftarrow trs_i$; $c^{\circ} \leftarrow c$ $\mathbf{G}_{\geq 2}$
39 $\quad e \leftarrow e + 1$ $\mathbf{G}_{\geq 2}$
40 $\quad \mathrm{T}[trs_i \| c] \leftarrow \mathrm{T}[trs_i] \| ((1, ck), c')$
$\quad\quad \| ((2, c), c'')$ $\mathbf{G}_{=3}$
41 $trs_i \xleftarrow{\shortparallel} c$
42 If $mr = \mathtt{F}$: $\mathrm{CK}_i[trs_i] \leftarrow k$
43 Return (pk_i, c)

Oracle $\mathrm{Solve}(i, tr, k)$
44 Require $1 \leq i \leq n$
45 Require $tr \notin \mathrm{XP}_i$
46 Require $\mathrm{CK}_i[tr] \neq \bot$
47 Reward $k = \mathrm{CK}_i[tr]$ $\mathbf{G}_{<1}$
48 If $i = n_{\text{Gen}}$: $\mathbf{G}_{\geq 1}$
49 $\quad$ If $tr = tr^{\circ} \| c^{\circ}$: $\mathbf{G}_{\geq 2}$
50 $\quad\quad$ Reward $k = \mathrm{CK}_i[tr]$ $\mathbf{G}_{\geq 1,2}$
51 Else if $k = \mathrm{CK}_i[tr]$: Abort $\mathbf{G}_{\geq 1}$
52 Return

Oracle $\mathrm{Up}_R(i, ad)$
53 Require $1 \leq i \leq n \wedge ad \in \mathcal{AD}$
54 $(st_A, st_B) \leftarrow \mathrm{SK}_i[trr_i]$
55 $(st_A, k, c) \leftarrow \mathrm{snd}(st_A, (0, ad); 0)$
56 $(st_B, k) \leftarrow \mathrm{rcv}(st_B, (0, ad), c)$
57 If $i = n_{\text{Gen}}$: $\mathbf{G}_{=3}$
58 $\quad \mathrm{T}[trr_i \| ad] \leftarrow \mathrm{T}[trr_i] \| ((0, ad), c)$ $\mathbf{G}_{=3}$
59 $trr_i \xleftarrow{\shortparallel} ad$
60 $\mathrm{SK}_i[trr_i] \leftarrow (st_A, st_B)$
61 Return

Oracle $\mathrm{Dec}(i, c)$
62 Require $1 \leq i \leq n \wedge c \in \mathcal{C}$
63 $(st_A, st_B) \leftarrow \mathrm{SK}_i[trr_i]$
64 $(ck, pk, c', \tau) \leftarrow c$
65 $(st_B, (k, k.m)) \leftarrow \mathrm{rcv}(st_B, (1, ck), c')$
66 If $trr_i = tr^{\circ}$
$\quad \wedge (ck^{\bullet}, c^{\bullet}) = (ck, c') \neq (\bot, \bot)$: $\mathbf{G}_{\geq 3}$
67 $\quad (k, k.m) \leftarrow (k^{\bullet}, k.m^{\bullet})$ $\mathbf{G}_{\geq 3}$
68 Require $\mathrm{vfy}_{\mathsf{M}}(k.m, (ck, pk, c'), \tau)$
69 If $trr_i = tr^{\circ} \wedge c \neq c^{\circ}$
$\quad \wedge (ck^{\bullet}, c^{\bullet}) = (ck, c') \neq (\bot, \bot)$: $\mathbf{G}_{\geq 4}$
70 $\quad$ Abort $\mathbf{G}_{\geq 4}$
71 $(st_A, k'', c'') \leftarrow \mathrm{snd}(pk, (2, c); 0)$
72 $(st_B, k''') \leftarrow \mathrm{rcv}(st_B, (2, c), c'')$
73 If $i = n_{\text{Gen}}$: $\mathbf{G}_{=3}$
74 $\quad \mathrm{T}[trr_i \| c] \leftarrow \mathrm{T}[trr_i] \| ((1, ck), c')$
$\quad\quad \| ((2, c), c'')$ $\mathbf{G}_{=3}$
75 $trr_i \xleftarrow{\shortparallel} c$
76 $\mathrm{SK}_i[trr_i] \leftarrow (st_A, st_B)$
77 If $\mathrm{CK}_i[trr_i] \neq \bot$:
78 $\quad$ Return
79 Return k

Oracle $\mathrm{Expose}(i, tr)$
80 Require $1 \leq i \leq n$
81 Require $\mathrm{SK}_i[tr] \in \mathcal{PK} \times \mathcal{SK}$
82 If $tr \preceq tr^{\circ}$: Abort $\mathbf{G}_{\geq 2}$
83 $\mathrm{XP}_i \xleftarrow{\cup} \{tr^* \in (\mathcal{AD} \cup \mathcal{C})^* : tr \prec tr^*\}$
84 $(st_A, st_B) \leftarrow \mathrm{SK}_i[tr]$
85 Return (st_A, st_B)

Figure 4.10: Games of simulation for proof of KUOWR security for construction from Figure 4.9.

4.5.2 One-Time MAC from URKE

Here we describe the construction of a one-time message authentication code from a generic URKE scheme. The symmetric MAC key k can be thought of as the concatenated random coins for both one state initialization and one invocation of the send algorithm of A in the URKE scheme. A tag is generated by taking the message as associated data for algorithm snd (which is invoked on randomness that is derived from key k). The MAC tag then contains the resulting ciphertext as well as the key output by algorithm snd. For verification, the rcv algorithm is invoked on the message as associated data and the ciphertext from the tag. The resulting key is then compared to the key from the MAC tag. We present this scheme in Figure 4.11.

Corollary 1 *If URKE scheme* UR *is* KINDR *secure, then one-time MAC scheme* M *from Figure 4.11 is* SUF *secure with* $\mathrm{Adv}^{\mathrm{suf}}_{\mathsf{M}}(\mathcal{A}) \leq q_{\mathrm{Gen}} q_{\mathrm{Vfy}} \mathrm{Adv}^{\mathrm{kindr}}_{\mathsf{UR}}(\mathcal{B})$ *where* q_{Gen} *is the number of* Gen *queries and* q_{Vfy} *is the number of* Vfy *queries in the multi-instance* SUF *notion.*

Please note that in the proof of Theorem 5, only one instance and one tag verification are necessary, resulting in a tight proof to URKE KINDR security.

The proof of single-instance SUF security with one Vfy oracle query is immediate: The Tag oracle of the SUF game is simulated by the KINDR's SndA and Challenge oracles to produce the MAC tag $\tau =$

Proc $\mathrm{tag}(k, m)$	**Proc** $\mathrm{vfy}_{\mathsf{M}}(k, m, \tau)$
00 $rc_i \| rc_s \leftarrow k$	05 $rc_i \| rc_s \leftarrow k$
01 $(st_A, st_B) \leftarrow \mathrm{init}(rc_i)$	06 $(st_A, st_B) \leftarrow \mathrm{init}(rc_i)$
02 $(st_A, \kappa, c) \leftarrow \mathrm{snd}(st_A, m; rc_s)$	07 $(\kappa, c) \leftarrow \tau$
03 $\tau \leftarrow (\kappa, c)$	08 $(st_B, \kappa') \leftarrow_{\$} \mathrm{rcv}(st_B, m, c)$
04 Return τ	09 Require $\kappa = \kappa'$
	10 Return $\mathtt{T}$

Figure 4.11: One-time MAC scheme M from generic URKE scheme $\mathsf{UR} = (\mathrm{init}, \mathrm{snd}, \mathrm{rcv})$.

(κ, c). When the (successful) adversary eventually provides a valid tag forgery $\tau^* = (\kappa^*, c^*)$ to the Vfy oracle, two cases are considered:

1. If $c \neq c^*$, then the KINDR's RcvB oracle is invoked on c^* and the Challenge oracle is queried on the resulting key. If this challenge key κ' equals κ^*, then $b' = 0$ is returned to the KINDR game. Otherwise $b' = 1$ is returned.

2. If $c = c^* \wedge \kappa \neq \kappa^*$, then $b' = 1$ is returned.

Please note that the reduction looses tightness linearly in the number of Gen and Vfy queries. As queries to Vfy either contain a known tag, an invalid tag, or a tag forgery, the simulation is straight forward: the reduction guesses, which unknown tag is the first correct tag forgery (and uses it to solve the KINDR game) and treats all remaining queries with unknown tags as invalid tags.

4.6 Discussion

Our results clearly show that key-updatable key encapsulation is a necessary building block for optimally secure ratcheted key exchange, if the security definition of the latter regards manipulation of the algorithm invocations' random coins. As unidirectional RKE can naturally be built from sesquidirectional RKE, which in turn can be built from bidirectional RKE (which can be derived from optimally secure group RKE), our results are expected to hold also for the according security definitions under these extended communication settings. In contrast, security definitions of ratcheting that restrict the adversary more than necessary in exposing the local state or in solving embedded game challenges (i.e., by excluding more than unpreventable attacks) allow for instantiations that can dispense with these inefficient building blocks.

However, the two previous security definitions fulfilled by constructions that use kuKEM as a building block (cf. Table 4.1) consider only *randomness reveal* [JS18a] or even *secure randomness* (sections 3.6 and 3.9). This raises the question whether using kuKEM in these

cases was indeed necessary (or not). The resulting gap between notions of ratcheting that can be instantiated from only standard PKC and our optimally secure URKE definition with *randomness manipulation*, implying kuKEM*, will be discussed in the following.

Implications under Randomness Reveal The core of our proof (showing that URKE implies kuKEM under randomness manipulation) is to utilize URKE's state update in algorithms snd and rcv for realizing public key and secret key updates in kuKEM's up algorithm. In order to remove the otherwise necessary communication between snd and rcv algorithms of URKE, snd is de-randomized by fixing its random coins to a static value. While this de-randomization *trick* is not immediately possible if the reduction to URKE KIND security cannot manipulate the randomness of snd invocations, one can utilize a programmable random oracle to emulate it: instead of fixing the (input) random coins of snd invocations to a static value, one could derive these coins from the output of a random oracle on input of the respective update's associated data (i.e., *ad* input of algorithm up). Additionally, instead of directly forwarding the update's associated data to the associated-data input of snd, another random oracle could be interposed between them. The reduction then simply pre-computes all kuKEM up invocations independent of associated-data inputs by querying the SndA oracle in the URKE KIND game on random associated-data strings. Then the reduction reveals all used random coins in the URKE KIND game and programs them as output into the random oracle lazily (i.e., as soon as the adversary queries the random oracle on update associated-data strings). By correctly guessing, which of the adversary's random oracle queries fit its queried kuKEM update invocations, the reduction can perform the same de-randomization trick as in our proof. The probability of guessing correctly is, however, exponential in the number of queried kuKEM updates such that a useful implication may only be derivable for a constant number of queried updates.

In conclusion, we conjecture that URKE under randomness re-

veal already requires the use of a kuKEM-like building block with a constantly bounded number of public key and secret key updates. Thereby we argue that our proof approach partially carries over to the case of randomness reveal. This would indicate that the use of a kuKEM-like building block in the construction of Jaeger and Stepanovs [JS18a] is indeed necessary. The formal analysis of this conjecture is left as an open question for future work.

Implications under Secure Randomness For optimal security under secure randomness, we show in sections 3.4 and 3.10 that URKE can be instantiated from standard PKC only (cf. Table 4.1). In contrast, our construction of sesquidirectional RKE from Section 3.6 uses kuKEM for satisfying optimal security under secure randomness. Since an adversary against SRKE (under KIND security with secure randomness) has no access to random coins respectively used in the RKE algorithms, our de-randomization trick seems inapplicable. Furthermore, while the RKE algorithms snd and rcv can use exchanged ciphertexts for their state updates, generically transforming this state update to realize a 'silent', non-interactive key update needed for kuKEM without our trick appears (at least) problematic.

Nevertheless, it is likely that SRKE KIND security under secure randomness requires kuKEM-like building blocks. This intuition is based on the example attack from Section 3.7.2 which we shortly recall and summarize. This attack illustrates that a key k^*, computed by any secure SRKE construction under the according adversarial execution, needs to be indistinguishable from a random key according to this security notion. The attack proceeds as follows: 1. Alice's and Bob's states are exposed ($st_A \leftarrow \mathrm{ExposeA}(\epsilon)$; $st_B \leftarrow \mathrm{ExposeB}(\epsilon)$), 2. Bob sends update information to Alice (which is possible in SRKE) to recover from his exposure ($c \leftarrow \mathrm{SndB}(\epsilon, \epsilon)$; $\mathrm{RcvA}(\epsilon, c)$). Keys established by Alice after receiving the update information are required to be secure again. Translated to the kuKEM setting, this step corresponds to Bob generating a new key pair and publishing the respective public key. 3. Simultaneously Alice is impersonated towards Bob ($(st'_A, k', c') \leftarrow_{\$}$

$\mathrm{snd}_A(st_A, \epsilon)$; $\mathrm{RcvB}(\epsilon, c')$). This requires Bob's state to become incompatible with Alice's state. In the kuKEM setting, this corresponds to the secret key being updated with c' as associated data. Note that c' can be independent of Bob's state update from step 2 via ciphertext c, and the computation of c' is fully controlled by the adversary. 4. Afterwards Bob's state is again exposed ($st'_B \leftarrow \mathrm{ExposeB}((\epsilon, c) \| (\epsilon, c'))$). 5. Finally, Alice sends and establishes key k^* which is required to be secure ($c'' \leftarrow \mathrm{SndA}(\epsilon, \epsilon)$). 6. Exposing Alice's state thereafter should not harm security of k^* ($st''_A \leftarrow \mathrm{ExposeA}((\epsilon, c''))$).

We observe that, as with a kuKEM public key, Alice's state is publicly known during the entire attack. Only Alice's random coins when establishing k^* and updating her state, and Bob's random coins when sending, as well as his resulting state until he receives c' are hidden towards the adversary. We furthermore note that, by computing ciphertext c', the adversary controls Bob's state update. As a consequence, Bob's state update must reach forward-secrecy for key k^* with respect to adversarially chosen associated update data c' and Bob's resulting (diverged) state st'_B.

All in all, the security requirements highlighted by this attack emphasize the similarity of kuKEM's and SRKE's security. Nevertheless, we note that all our attempts to apply our proof technique for this case failed due to the above mentioned problems. Therefore, formally substantiating or disproving the intuition conveyed by this attack remains an open question for future work.

Open Questions and Impact With the results in this chapter we aim to motivate research on another remaining open problem: can key-updatable KEM be instantiated more efficiently than generically from HIBE? It is, in contrast, evident that equivalence between HIBE and RKE is unlikely as constructions of the latter only utilize one 'identity path' of the whole 'identity tree' of the former.

Conclusively, we note that defining security for, and constructing schemes of interactive ratcheted key exchange variants (i.e., under bidirectional communication) is highly complicated and consequently

error-prone.[5] By providing generic constructions (instead of ad-hoc designs) and grasping core components and concepts of ratcheted key exchange, complexity is reduced and sources of errors are eliminated. Additionally, our equivalence result serves as a benchmark for current and future designs of ratcheted key exchange—especially group RKE. For future constructions that only rely on standard public key cryptography either of the following questions may arise: how far is the adversary restricted such that our implication is circumvented, or how far is the construction secure under the respective security definition?

5

Communication Costs of Ratcheting in Groups

Contents

5.1 Introduction 168
5.2 Security of Concurrent Group Ratcheting 176
5.3 Deficiencies of Existing Protocols 179
5.4 Key-Updatable Public Key Encryption 185
5.5 Intuition for Lower Bound 186
5.6 Upper Bound of Communication Complexity 194
5.7 Lower Bound of Communication Complexity 202
5.8 Discussion 227

While ratcheting in the two-party setting is the focus of the previous chapters and has generally attracted a lot of attention in the literature, the problem of recovering from the exposure of honest protocol participants' local states, also called *post-compromise security* (PCS), in the group setting (i.e., *group ratcheting*) is much less understood.

On the one hand, one can achieve excellent security by simply executing, in parallel, a two-party ratcheting protocol (e.g., our BRKE construction from Chapter 3 or less secure but more efficient variants like the Signal protocol) for each pair of members in a group. However, this incurs $\mathcal{O}(n)$ communication overhead for every message sent, where n is the group size. On the other hand, several related protocols were recently developed in the context of the IETF Messaging Layer Security (MLS) effort that improve the communication overhead per message to $\mathcal{O}(\log n)$. However, this reduction of communication overhead involves a great restriction: group members are not allowed to

send and recover from exposures concurrently such that reaching PCS is delayed up to n communication time slots (potentially even more).

In this chapter we formally study the trade-off between PCS, concurrency, and communication overhead in the context of group ratcheting. Since our main result here is a lower bound, we define the cleanest and most restrictive setting where the tension already occurs: *static* groups equipped with a *synchronous* (and authenticated) broadcast channel, where up to t arbitrary parties can concurrently send messages in any given round. Already in this setting, we show in a symbolic execution model that PCS requires $\Omega(t)$ communication overhead per message. Our symbolic model permits as building blocks black-box use of (even dual) PRFs, (even key-updatable) PKE (which in our symbolic definition is at least as strong as HIBE), and broadcast encryption, covering all tools used in previous group ratcheting constructions, but prohibiting the use of exotic primitives.

To complement our result, we also prove an almost matching upper bound of $\mathcal{O}(t \cdot (1 + \log(n/t)))$, which smoothly increases from $\mathcal{O}(\log n)$ with no concurrency, to $\mathcal{O}(n)$ with unbounded concurrency, matching the previously known protocols.

Contributions by the Author This entire chapter has almost exclusively been contributed by the author of this thesis. The full version [BDR20c] of the paper that has been published in the proceedings of TCC 2020 [BDR20b] contains, in addition to the contents of this chapter, both a formal proof of performance and security for the upper-bound construction from Section 5.6. Sections 5.1 and 5.6 were jointly contributed by the author of this thesis and co-authors from [BDR20b].

5.1 Introduction

We recall that *post-compromise security* [CCG16] refers to the ability of a given protocol to recover—by means of normal protocol operations —from the exposure of local states of its (otherwise honest)

participants.

The concepts behind realizing PCS in group ratcheting can be split into two approaches: On the one extreme, several systems, including Signal Messenger itself, achieve PCS in groups by simply executing, in parallel, a two-party PCS-secure protocol (e.g., our BRKE construction or the Signal protocol) for each pair of members in a group. In addition to achieving PCS, this simple technique is also extremely resilient to asynchronicity and concurrency: people can send messages concurrently, receive them out-of-order, or be off-line for extended periods of time. However, it comes at a steep communication overhead $\mathcal{O}(n)$ for every message sent, where n is the group size.

On the other hand, several related protocols [CCG+18, ACDT20, ACC+19a, ACJM20] (some of them introduced under the term *continuous group key agreement* (CGKA)[1]) were recently developed in the context of the IETF Message Layer Security (MLS) initiative for group messaging [BBM+20a]. One of the main goals of this initiative was to achieve PCS in groups with a significantly lower communication overhead. And, indeed, for static groups, these protocols improve this overhead per message to $\mathcal{O}(\log n)$, More precisely, these protocols separate protocol messages into two categories: *Payload* ciphertext, used to actually encrypt messages, have no overhead, but also do not help in establishing PCS. In contrast, *update* ciphertexts carry no payload, but exclusively establish PCS: intuitively, an update ciphertext from user A refreshes all cryptographic material held by A. These update ciphertexts have size proportional to $\mathcal{O}(\log n)$ in MLS-related protocols, which is a significant saving for large groups, compared to the pairwise-Signal protocol.

Concurrency. Unfortunately, this reduction of communication overhead for MLS-related protocols involves a great restriction: *all update ciphertexts must be generated and processed one-by-one in the same*

[1] By distinguishing between 'CGKA' and 'group ratcheting', these works differentiate between the asymmetric cryptographic parts of the protocols and the entire key establishment procedure, respectively [ACJM20]. In order to avoid this strict distinction, we call it 'group ratcheting' here.

order by all the group members. We stress that this does not just mean that update ciphertexts can be prepared concurrently, but processed in some fixed order. Instead, a fresh update ciphertext cannot be *prepared* until all previous update ciphertexts are *processed.* In particular, it is critical to somehow implement what these protocols call a 'delivery server', whose task is to reject all-but-one of the concurrently prepared update ciphertexts, and then to ensure that all group members process the 'accepted updates' in the same correct order. Implementing such a delivery server poses a significant burden not only in terms of usability (which is clear), but also for *security* of these protocols, as it delays reaching PCS up to n communication time slots (potentially more in asynchronous settings, such as messaging). Indeed, the concurrency restriction of MLS is currently one of the biggest criticisms and hurdles towards its wide-spread use and adoption (see [ACDT20] for extensive discussion of this). In contrast, pairwise Signal does not have any such concurrency restriction, albeit with a much higher communication overhead. See Section 5.3 and Table 5.1 for more detailed comparison of various existing methods for group ratcheting.

Our Main Question. This brings us to the main question we study in this chapter:

What is the trade-off between PCS, concurrent sending and low communication complexity in group ratcheting protocols?

For our lower bound, we define the cleanest and most restrictive setting where the tension already occurs: *static* groups equipped with a *synchronous* (and authenticated) broadcast channel, where up to t arbitrary users can concurrently send ciphertexts in any given round. In particular, $t = 1$ corresponds to the restrictive MLS setting which, we term 'no concurrency', and $t = n$ corresponds to unrestricted setting achieved by pairwise Signal, which we term 'full concurrency'. Also, without loss of generality, and following the convention from previous chapters that is also established in MLS-related protocols, we focus on the 'key encapsulation' mechanism of group ratcheting protocols. Namely, our model is the following:

We have a static group of n members whose goal is to continuously share a group key k. Group members have private states st, and communicate in rounds over a public broadcast channel. Each round refreshes the current group key k into the next group key k' as follows: 1. At the beginning of a round, an arbitrary subset of up to t group members is selected by the adversary to update the current group key k. These group members are called *senders* (of a given round). 2. During each round, each sender—*unaware of the identities of other senders*—tosses fresh random coins, sends a ciphertext c over the broadcast channel, and updates its private state st. 3. At the end of each round, all (up to t) ciphertexts c are received by all n users, who use them to update their state st, and output a new group key k'. 4. At the end of each round, the adversary can learn the current group key k', and is also allowed to expose an arbitrary number of group member states st.

For our lower bound, we will demand the following, rather weak, PCS guarantee. A key k after round i (not directly revealed to the attacker) is secure if: (a) no user is exposed in round $i' \geq i$; (b) all users sent at least one update ciphertext between their latest exposure and round $i-1$; and (c) after all exposed users sent once without being exposed again, at least one user additionally sent in round $j \leq i$. Condition (a) will only be used in our lower bound (to make it stronger), to ensure that our lower bound is only due to PCS, but not the complementary property *forward*-secrecy, which states that past round keys cannot be compromised upon current state exposure. However, our upper bound will achieve forward-secrecy, dropping (a).

Condition (b) is the heart of PCS, demanding that security should be eventually restored once every exposed user updated its state. Condition (c) permits a one-round delay before PCS takes place. While not theoretically needed, avoiding this extra round seems to require some sort of multiparty non-interactive key exchange for *concurrent* state updates, which currently requires exotic cryptographic assumptions, such as multi-linear maps [BGK+18, BS02]. In contrast, the extra round allows to use traditional public-key cryptography techniques, such as the exposed user sending fresh public-keys, and future

senders using these keys in the extra round to send fresh secret(s) to this user. While condition (c) strengthens our lower bound, our upper bound construction can be minimally adjusted to achieve PCS for *non-concurrent* state updates even without this 'extra round'.

Our Upper Bound. We show nearly matching lower and upper bounds on the efficiency of t-concurrent, PCS-secure group ratcheting schemes. With our upper bound we provide a group ratcheting scheme with communication overhead $\mathcal{O}(t \cdot (1 + \log(n/t)))$, which smoothly increases from $\mathcal{O}(\log n)$ with no concurrency, to $\mathcal{O}(n)$ with unbounded concurrency, matching the upper bounds of the previously known protocols. Our upper bound is proven in the standard computational model. For the weak notion of PCS alone sketched above (i.e., conditions (a)-(c)), we only need public-key encryption (PKE) and pseudo-random functions (PRFs). Our construction carefully borrows elements from the complete subtree method of [NNL01] used in the context of broadcast encryption (BE), and the TreeKEM protocol of the MLS standard [BBM+20a, ACDT20] used in the context of non-concurrent group ratcheting. Similarly, one can view our construction as an adapted combination of components from Tainted TreeKEM [ACC+19a] and the most recent MLS draft [BBM+20b] with its propose-then-commit technique. By itself, none of these constructions is enough to do what we want: BE scheme of [NNL01] allows to send a fresh secret to all-but-t senders from the previous round (this is needed for PCS), but needs centralized distribution of correlated secret keys to various users, while the TreeKEM schemes no longer need a group manager, but do not withstand concurrency of updates in a rather critical way. Finally, the propose-then-commit technique, when naively combined with (Tainted) TreeKEM as in MLS [BBM+20b], in the worst case induces an overhead linear in the group size, and still does not completely achieve our desired concurrency and PCS guarantees. Nevertheless, we show how to combine these structures together—in a very concrete and non-black-box way—to obtain our scheme with overhead $\mathcal{O}(t \cdot (1 + \log(n/t)))$.

Moreover, we can easily achieve forward-security in addition to PCS

(i.e., drop restriction (a) on the attacker), by using the recent technique of [JMM19, ACDT20], which basically replaces traditional PKE with so called *updatable* PKE (uPKE). Informally, such PKE is *stateful*, and only works if all the senders are synchronized with the recipient (which can be enforced in our model, even with concurrency). Intuitively, each uPKE ciphertext updates the public and secret keys in a correlated way, so that future ciphertexts (produced with new public key) can be decrypted with the new secret key, but old ciphertexts cannot be decrypted with the new secret key. Hence, uPKE provides an efficient and practical mechanism for forward-secrecy in such a synchronized setting, without the need of heavy, less efficient but more powerful tools, such as hierarchical identity based encryption (HIBE), directly used as a building block for strongly secure group ratcheting [ACJM20], or used as an intermediary component to build stronger *key-updatable* PKE (kuPKE)[2] as (similarly) introduced in sections 3.2 and 4.2.

Our Lower Bound. We prove a lower bound $\Omega(t)$ on the efficiency of any group ratcheting protocol which only uses 'realistic' tools, such as (possibly key-updatable[2]) PKE, (possibly so called *dual*) PRFs, and general BE (see Chapter 2 for explanations of these terms). We define our symbolic notion of key-updatable PKE so that it even captures functionality and security guarantees at least as strong as one expects from HIBE. To the best of our knowledge, these primitives include all known tools used in all 'practical' results on group ratcheting (including our upper bound). Thus, our result nearly matches our upper bound, and shows that the $\mathcal{O}(n)$ *overheard of pairwise Signal protocol is optimal for unbounded concurrency*, at least within our model.

To motivate our model for the lower bound, group ratcheting would be 'easy' if we could use 'exotic' tools, such as multiparty non-interactive key agreement (mNIKE), multi-linear maps, or general-purpose obfus-

[2]While for our upper bound construction weaker and more efficient uPKE (based on DH groups) suffices as in [JMM19, ACDT20], to strengthen our lower bound we allow constructions to use stronger and less efficient key-updatable PKE (thus far based on HIBE) as (similarly) introduced in sections 3.2 and 4.2.

cation. For example, using general mNIKE, one can easily achieve PCS and unbounded concurrency, by having each member simply broadcast its new public key, without any knowledge of other senders: at the end of each round, the union of latest keys of all the group members magically (and non-interactively) updates the previous group key to a new, unrelated value. Of course, we currently do not have any even remotely practical mNIKE protocols, so it seems natural that we must define a model which only permits the use of 'realistic' tools, such as (ku)PKE, (dual) PRFs, BE, (HIBE,) etc.

To formally address this challenge, we use a *symbolic* modeling framework inspired by the elegant work of Micciancio and Panjwani [MP04], who used it to derive a lower bound for the efficiency of multicast encryption. Symbolic models treat all elements as symbols whose algebraic structure is entirely disregarded, and which can be used only as intended. For example, a symbolic public key can be defined to only encrypt messages, and the only way to decrypt the resulting ciphertext is to have another symbol corresponding to the associated secret key. In particular, under this definition one cannot perform any other operations with the symbolic public key, such as verifying a signature, using it for a Diffie–Hellman key exchange, etc.

We use such a symbolic model to precisely define the primitives we allow, including the grammar of symbols and valid derivation rules between them. We then formalize the intuition for our lower bound in Section 5.5 (before doing a formal proof in Section 5.7). Our bound is actually very strong: it is the *best-case* lower bound, which holds for any execution schedule of group ratcheting protocols within our model, and which is proven against highly restricted adversaries for extremely little security requirements. Specifically, we show that each sender for round i must send at least one fresh ciphertext over the broadcast channel 'specific' to every sender of the previous round $(i-1)$.[3] While intuitively simple, the exact formalization of this result is non-trivial, in part due to the rather advanced nature of the underlying primitives

[3]Except for itself, if the sender was active in the prior round. This intuitively explains why our 'best-case' lower bound is actually $(t-1)$ and not t.

we allow. For example, we must show that no matter what shared infrastructure was established before round $(i-1)$, and no matter what information a sender A sent in round $(i-1)$, there is no way for A to always recover at round i from potential exposure at round $(i-2)$, unless every sender B in round i sends some ciphertext 'only to A'.

Perspective. To put our symbolic result in perspective, early use of symbolic models in cryptography date to the Dolev-Yao model [DY81], and were used to prove 'upper bounds', meaning security of protocols which were too complex to analyze in the standard *computational model* (with reductions to well established simpler primitives or assumptions). In contrast, Micciancio and Panjwani [MP04] observed that symbolic models can also be used in a different way to prove impossibility results (i.e., *lower* bounds) on the efficiency of building various primitives using a fixed set of (symbolic) building blocks. This is interesting because we do not have many other compelling techniques to prove such lower bounds.

To the best of our knowledge, the only other technique we know is that of *black-box separations* [IR89]. While originally used for black-box impossibility results [IR89], Gennaro and Trevisan [GT00] adapted this technique to proving efficiency limitations of black-box reductions, such as building psedorandom generators from one-way permutations. However, black-box separation lower bounds are not only complex (which to some extent is true for symbolic lower bounds as well), but also become exponentially harder, as the primitive in question becomes more complex to define, or more diverse building blocks are allowed. In particular, to the best of our knowledge, the setting of group ratcheting using kuPKE, HIBE, dual PRFs, and BE used in this chapter, appears several orders of magnitude more complex than what can be done with the state-of-the-art black-box lower bounds.

Thus, we hope that the results in this chapter renew the interests in symbolic lower bounds, and that our techniques would prove useful to study other settings where such lower bounds could be proven.

5.2 Security of Concurrent Group Ratcheting

In this chapter we consider an abstraction of group ratcheting under significant relaxations and restrictions with respect to the real-world. The purpose of this approach is to disregard irrelevant aspects in order to highlight the immediate effects of concurrent state updates in group ratcheting.

In the following, we define syntax and (restricted) security of ratcheting in static groups against computationally bounded adversaries. We assume in our model that all group members have access to a round-based reliable and authenticated broadcast. Additionally, since our focus are concurrent operations in an initialized group, we consider an abstract initialization algorithm for deriving initial user states.[4]

Syntax A static group ratcheting protocol for a finite key space $\mathcal{K}$ is a triple $\mathsf{GR} = (\text{init}, \text{snd}, \text{rcv})$ of algorithms together with a space of local states $\mathcal{ST}$, a ciphertext space $\mathcal{C}$, and a space of random coins $\mathcal{R}$. The initialization algorithm init takes as input the number of group members $n \in \mathbb{N}$ and (freshly sampled) random coins $r \in \mathcal{R}$, and creates an initial local state $st_i \in \mathcal{ST}$ for every participating group member $i \in [n]$. For sending, a member invokes algorithm snd with its own current local state $st \in \mathcal{ST}$ and random coins $r \in \mathcal{R}$ to obtain as output its updated state $st' \in \mathcal{ST}$ and update information within a ciphertext $c \in \mathcal{C}$ that is to be sent via the broadcast. The receive algorithm rcv takes the member's current state $st \in \mathcal{ST}$ and a set of update ciphertexts $\boldsymbol{c} \subset \mathcal{C}$ (e.g., all broadcast ciphertexts since this member's last receiving), and outputs the updated state $st' \in \mathcal{ST}$ and the current (common) group key $k \in \mathcal{K}$, or a rejection symbol. As a consequence, all members compute the current group key on receipt (and not already when sending). A shortcut notation for these

[4] We note that we only consider a single independently established group session. For protocols in which participants use the same secrets simultaneously across multiple (thereby dependent) sessions, we refer the reader to a work by Cremers et al. [CHK19]. Both the problems and the solutions for these two considerations appear to be entirely distinct.

algorithms is

$$\begin{array}{rcccl} \mathbb{N} \times \mathcal{R} & \to & \text{init} & \to & \mathcal{ST}^n,\ n \in \mathbb{N} \\ \mathcal{ST} \times \mathcal{R} & \to & \text{snd} & \to & \mathcal{ST} \times \mathcal{C} \\ \mathcal{ST} \times \mathcal{P}(\mathcal{C}) & \to & \text{rcv} & \to & (\mathcal{ST} \times \mathcal{K}) \cup \{(\bot, \bot)\} \end{array}$$

For the treatment in our symbolic model, we consider (secret) randomness explicitly.

Security Security experiments $\text{KIND}^b_{\mathsf{GR}}$, formally defined in Figure 5.1, in which adversary $\mathcal{A}$ attacks scheme GR proceed as follows: Adversary $\mathcal{A}$ first determines the number of group members n. Afterwards the challenger invokes the init algorithm to generate initial secret states for all members. Then the security experiment continues in rounds. In every round i

- adversary $\mathcal{A}$ chooses set $\boldsymbol{U}^i_{\mathbf{S}}$ of senders. For each sender $u \in \boldsymbol{U}^i_{\mathbf{S}}$ algorithm snd is invoked. All resulting ciphertexts are both given to $\mathcal{A}$ and received by all group members via invocations of algorithm rcv.
- adversary $\mathcal{A}$ chooses set $\boldsymbol{U}^i_{\mathbf{X}}$ of exposed users. The local state of each user $u \in \boldsymbol{U}^i_{\mathbf{X}}$ after receiving in round i is given to $\mathcal{A}$.

During the entire security experiment, $\mathcal{A}$ can challenge group keys established in any round i^*. $\mathcal{A}$ either obtains a random key (if $b = 0$) or the actual group key from round i^* (if $b = 1$) in response. When terminating, $\mathcal{A}$ returns a guess b' such that it wins if $b = b'$ and for all challenged group keys it holds that:

(a) no user was exposed after a challenged group key was computed (see Figure 5.1 line 30),
(b) every user sent at least once after being exposed and before a challenged group key was computed (see line 20), and
(c) after all exposed users sent once without being exposed again, at least one user additionally sent before a challenged group key was computed (see line 20).

Group keys for which conditions (a)-(c) hold are marked *secure*.

We restrict the adversary with condition (a) only because the resulting weaker security definition already suffices to prove our *lower*

Game $\mathrm{KIND}^b_{\mathsf{GR}}(\mathcal{A})$
00 $phase \leftarrow 1$; $i \leftarrow 1$
01 $\mathrm{K}[\cdot] \leftarrow \bot$; $\boldsymbol{XU} \leftarrow \emptyset$
02 $\boldsymbol{SEC} \leftarrow \emptyset$; $\boldsymbol{CH} \leftarrow \emptyset$
03 $(\varsigma, n) \leftarrow_{\$} \mathcal{A}$
04 $(st_1, \ldots, st_n) \leftarrow_{\$} \mathrm{init}(n)$
05 $phase \leftarrow 2$
06 $b' \leftarrow_{\$} \mathcal{A}(\varsigma)$
07 If $\boldsymbol{CH} \subseteq \boldsymbol{SEC}$:
08 Stop with b'
09 Stop with 0

Oracle Reveal(i^*)
10 Require $\mathrm{K}[i^*] \neq \bot$
11 $k \leftarrow \mathrm{K}[i^*]$; $\mathrm{K}[i^*] \leftarrow \bot$
12 Return k

Oracle Round($\boldsymbol{U}$)
13 Require $phase > 1$
14 Require $\boldsymbol{U} \subseteq [n]$
15 For all $u \in \boldsymbol{U}$:
16 $(st_u, c_u) \leftarrow_{\$} \mathrm{snd}(st_u)$
17 $\boldsymbol{C} \leftarrow \bigcup_{u \in \boldsymbol{U}} c_u$
18 For all $u \in [n]$:
19 $(st_u, k_u) \leftarrow \mathrm{rcv}(st_u, \boldsymbol{C})$
20 If $\boldsymbol{XU} = \emptyset \wedge (\boldsymbol{U} \neq \emptyset$
$\vee i - 1 \in \boldsymbol{SEC})$:
21 $\boldsymbol{SEC} \leftarrow \{i\}$
22 $\boldsymbol{XU} \leftarrow \boldsymbol{XU} \setminus \boldsymbol{U}$
23 $\mathrm{K}[i] \leftarrow k_1$
24 $i \leftarrow i + 1$
25 Return $\boldsymbol{C}$

Oracle Expose($\boldsymbol{U}$)
26 Require $phase > 1$
27 Require $\boldsymbol{U} \subseteq [n]$
28 $\boldsymbol{XU} \leftarrow \boldsymbol{XU} \cup \boldsymbol{U}$
29 $\boldsymbol{ST} \leftarrow \bigcup_{u \in \boldsymbol{U}} \{st_u\}$
30 $\boldsymbol{SEC} \leftarrow \boldsymbol{SEC} \setminus [i-1]$
31 Return $\boldsymbol{ST}$

Oracle Challenge(i^*)
32 Require $\mathrm{K}[i^*] \neq \bot$
33 $\boldsymbol{CH} \leftarrow \boldsymbol{CH} \cup \{i^*\}$
34 $k^0 \leftarrow_{\$} \mathcal{K}$
35 $k^1 \leftarrow \mathrm{K}[i^*]$
36 $\mathrm{K}[i^*] \leftarrow \bot$
37 Return k^b

Figure 5.1: Security experiment of concurrent group ratcheting in the computational model. Note that the overall game mechanism corresponds to the one in the symbolic setting (see Figure 5.7) except that we require key indistinguishability here. Line 30 is removed for our construction's security to require immediate forward-secrecy.

bound of communication complexity. For our full model in which our construction for the *upper* bound is secure, we strengthen adversaries by lifting restriction (a) by ignoring line 30 in Figure 5.1. This reflects that our upper bound construction achieves immediate forward-secrecy while our lower bound already holds without requiring any form of forward-secrecy.

Condition (b) models that a user who was exposed must generate fresh secrets and send the respective public values to the group before it can receive confidential information for establishing new secure group keys. After all exposed users recovered by sending subsequently, their sent contribution must be used effectively to establish a new secret group key. Therefore, condition (c) additionally requires one further response from a user as a reaction to all newly contributed public values.

For removing condition (c) either 1. the last users who recovered did so concurrently at most as a pair of two (such that their new public

contributions can be merged into a shared group key non-interactively with NIKE mechanisms), or 2. multiparty NIKE schemes exist (for resolving cases of more concurrently recovering users). In order to simplify our security definition by not introducing an according case distinction tracing occurrences of case 1, we generally restrict the adversary with condition (c). We note that for proving our lower bound, restricting the adversary by this condition strengthens our result.

Intuitively, a group ratcheting scheme is secure if no adversary $\mathcal{A}$ exists that wins the above defined security experiment with probability non-negligibly higher than 1/2.

Restrictions of the Model With the following abstractions, simplifications, and restrictions in our above defined model, we support clarity and comprehensibility of our results and strengthen the statement of our lower bound. We consider: 1. A round-based communication setting, 2. Static groups, 3. All group members receive in every round, 4. Only passive adversaries 5. Adversaries can expose users only after receiving, and 6. Adversaries cannot attack used randomness. As we do not aim to develop a functional and secure group messenger but to theoretically analyze the foundations of concurrent group ratcheting, we believe this is justified.

5.3 Deficiencies of Existing Protocols

The problem of constructing group ratcheting could be solved trivially if efficient *multiparty non-interactive key exchange* schemes existed. Especially for the concurrent recovery from state exposures in group ratcheting, the lack of this tool appears to be crucial: Due to not being able to combine independently proposed fresh public key material, existing efficient group ratcheting constructions cannot process concurrent operations as we will explain in this section. In Table 5.1 we summarize the characteristics of previous group ratcheting schemes in comparison to our construction and the lower bound.

	PCS	Concurrency	Overhead
Sender Key Mechanism [RMS18]	○	●	1
Parallel Signal [RMS18, CCD+17, ACD19]	●	●	n
Asynchronous Ratcheting Trees [CCG+18]	●	○	$\log(n)$
Causal TreeKEM [Wei19]	◐	◑	$\log(n)$
TreeKEM Familiy [ACDT20, ACC+19a]	●	○	$\log(n)$
MLS Draft-09 [BBM+20b]	◐	◑	n
Strengthened Tainted TreeKEM [ACJM20]	◐	◑	$\log(n)$
Our Construction	●	●	$t \cdot (1 + \log(n/t))$
Our Lower Bound	●	●	$t - 1$

Table 5.1: Properties of group ratcheting constructions and our lower bound. $t = |\boldsymbol{U}_{\mathbf{S}}^{i-1}|$ is the number of members who sent concurrently in the previous round. For the overhead we consider a worst-case scenario in a constant size group. Constructions denoted with '◐'/'◑' provide PCS under no concurrency and can handle concurrent state updates without reaching PCS with them.

Sender Key Mechanism WhatsApp uses the so called *sender key mechanism* for implementing group chats [RMS18]. This mechanism distributes a symmetric *sender key* for each member in a group. When sending a group message, the sender protects the payload with its own sender key, transmits the resulting (single) ciphertext, and hashes the used sender key to obtain its next sender key. The receivers decrypt the ciphertext with the sender's sender key and also update the sender's sender key by hashing it.

While the deterministic derivation of sender keys induces no communication overhead after the initial distribution of sender keys, it implies the reveal of all future sender keys as soon as a member state is exposed (breaking PCS). However, as each group member's key material is processed and used independently, concurrently initiated group operations can be processed naturally.

Parallel Execution of Pairwise Signal The group ratcheting mechanism implemented in the Signal messenger bases on parallel executions of the two-party Double Ratchet Algorithm [PM16, CCD+17, ACD19] between each pair of members in a group [RMS18]. Due to splitting the group of size n into its n^2 independent pairwise components, this construction can naturally handle concurrency. At the same time, this approach induces a communication overhead of $\mathcal{O}(n)$

ciphertexts per sent group payload.

Since the Double Ratchet Algorithm reaches PCS for each pair of members, also its parallel execution achieves this goal for the group against passive adversaries or if the member set remains static. In an independent work [Rös18, RMS18] we describe an active attack against PCS in dynamic groups that exploits the decentralized membership management implemented in Signal. Furthermore, the delayed recovery from state exposures in the Double Ratchet Algorithm due to a strictly alternating update schedule between protocol participants (cf. analysis and fix in [ACD19]) lets recoveries from state exposures in the group become effective only after every group member sent once at worst. With stronger two-party ratcheting protocols (e.g., our constructions from Chapter 3 or [JS18a, ACD19, JMM19]) this problem can be solved.

Asynchronous Ratcheting Tree While the two above described approaches compute and use multiple symmetric keys in parallel for protecting communication in groups, the following constructions do so by deriving a single shared group key at each step of the group's lifetime. Therefore they arrange asymmetric key material on nodes in a tree structure in which each leaf represents a group member and the common root represents the shared group secret. Every group member stores the asymmetric secrets on the path from its leaf to the common root in its local state. For updating the local state, in order to recover from an adversarial exposure, all constructions let the updating member generate new asymmetric secrets for each node on their path to the root.

In the Asynchronous Ratcheting Trees (ART) design [CCG+18], these asymmetric secrets are exponents in a Diffie–Hellman (DH) group. State updates of a member's path is conducted as follows: the updating member freshly samples a new secret exponent for its own leaf and then deterministically derives every ancestor node's secret exponent as the shared DH key from its two children's public DH shares. All resulting new public DH shares on the path are sent to the

group, inducing a communication overhead of $\mathcal{O}(\log(n))$ per update operation. Other members perform the same derivations for updated nodes on their own paths to the root to obtain the new exponents. Since all secrets in the updating member's local state are renewed based on fresh random coins, this mechanism achieves PCS.

The reason for ART not being able to process concurrent update operations is that simultaneous updates of nodes in the tree with independently computed DH exponents cannot be merged into a joint tree structure while reaching PCS. For t concurrent updates, a t-party NIKE would be needed to combine the resulting t new proposed DH shares into a shared secret exponent for the ancestor node at which all updating members' paths to the root join together. (As mentioned before, if multiparty NIKE existed, group ratcheting can be solved trivially without complex tree structures.)

Causal TreeKEM As in the ART design, Causal TreeKEM [Wei19] uses exponents in a DH group as asymmetric secrets on nodes in the tree. Also the update procedure is conceptually the same. However, in case of concurrently proposed path updates, the conflicting new exponents on a node are combined via exponent-addition and the conflicting public DH shares on a node are combined via multiplying these group elements.

Although this merge-mechanism resolves conflicts caused by concurrency, the combination of updated path secrets is not post-compromise secure: the old exponents of two nodes (from which their updating users A and B aimed to recover), whose common parent was updated via a combination of concurrent path updates, suffice to derive their parent's resulting new exponent. (The new exponent is the old exponent mixed with random values from A and B that they encrypt to the other's old node key.)

TreeKEM Family The family of TreeKEM protocols [ACDT20, ACC+19a] uses as asymmetric key material for nodes in the tree key encapsulation mechanism (KEM) key pairs or, in forward-secure

TreeKEM, updatable KEM key pairs. For updating its local state, a group member samples a fresh secret from which it deterministically derives seeds for each node on its path to the root, such that all ancestor seeds can be derived from their descendant seeds (but not vice versa). The updating member generates the new key pair for each updated node from its seed deterministically, and encapsulates the node's seed to the public key of the child which is not on the member's path to the root. This mechanism achieves PCS and induces a communication overhead of $\mathcal{O}(\log(n))$ per update.

The idea of recovery from exposures is undermined in case of concurrency, since updating members send their new seeds for a node on their path to public keys of siblings, simultaneously being updated and replaced by new key material of members who concurrently update: the potentially exposed secrets *from which* one updating member aims to recover can then be used to obtain the new secrets *with which* the other updating user aims to recover (as in the case of Causal TreeKEM). Consequently, concurrent updates in TreeKEM are essentially ineffective with respect to PCS.

Forward-secure TreeKEM [ACDT20] uses an updatable KEM for enhancing forward-security guarantees of the above described mechanism. Tainted TreeKEM [ACC+19a] enhances PCS guarantees with respect to dynamic membership changes in groups. Neither of these changes affect the trade-offs discussed here.

MLS Draft-09 Based on TreeKEM, the most recent draft of MLS [BBM+20b] distinguishes between two state update variants: (a) In an *update proposal* a member refreshes only its own leaf key pair, removes all other nodes on the path from this leaf to the root, and makes the root parent of all nodes that thereby became parentless. (b) In a *commit* a member combines previous update proposals and refreshes all key pairs on the path from its own leaf to the root (matching the normal TreeKEM update as described in the last paragraph).

In principle, both update variants achieve PCS for respective the sender. However, for simultaneously sent *commits*, all but one are re-

jected (e.g., by a central server) meaning that PCS under concurrency is not achieved for rejected updating commits. Furthermore, while *update proposals* can be processed concurrently, they eventually let the tree's depth degrade to 1, inducing a worst-case overhead of $\mathcal{O}(n)$ for later commits.[5]

Optimally Secure Tainted TreeKEM Recently and concurrent to our work for this chapter, an optimally secure variant of group ratcheting, based on a combination of Tainted TreeKEM and MLS draft-09, was proposed by Alwen et al. [ACJM20]. In addition to authentication guarantees (which is independent of our focus), their protocol achieves strong security guarantees for group partitions caused by concurrency: instead of assuming that a (consensus) mechanism rejects conflicting commits as in MLS, they anticipate that different sub-groups of group members may process different of these commits such that the overall perspective on the group diverges. Their protocol guarantees that, after diverging, exposing states of one sub-group's members does not affect the security of another sub-groups' secrets. Intuitively, this is achieved by using HIBE key pairs on the tree's nodes that are regularly updated via secret-key-delegation based on identity strings that reflect the current perspective on the group. (For details, we refer the interested reader to [ACJM20].)

While these changes increase security with respect to some form of forward-secrecy under group partitions, they do not entirely solve the issue of conflicting commits as in MLS: committed state updates still only have an effect in a sub-group that processes the commit such that only one user at a time can update secrets on the path from its leave to the root whereas other user's path updates remain ineffective.

Our construction from Section 5.6 bypasses the issue of concurrently generated, incompatible path proposals by postponing the update of affected nodes in the tree by one communication round. However, 'im-

[5] Consider, for example, a scenario in which the same majority of members always sends update proposals and a fixed disjoint set of few members always commits. In this case, the overhead of commits for these few members converges to $\mathcal{O}(n)$.

mediate' PCS can still be reached for non-concurrent updates by composing our construction with one of the above described ones without loss in efficiency. We note that some of the above constructions provide strong security guarantees with respect to active adversaries, dynamic groups, entirely asynchronous communication, or weak randomness, which is out (and partially independent) of our consideration's scope.

5.4 Key-Updatable Public Key Encryption

Before turning to our results, we introduce the notion of key-updatable public key encryption, used in this chapter. As key-updatable KEM (introduced in Sections 3.2 and 4.2) adds an update mechanism for the key pair components to standard KEM, key-updatable public key encryption (kuPKE) is an extension of standard public key encryption that allows for independent updates of public and secret key with respect to associated data.

A kuPKE scheme for a message space $\mathcal{M}$ and an associated data space $\mathcal{AD}$ is a quadruple $\mathsf{UE} = (\text{gen}, \text{up}, \text{enc}, \text{dec})$ of algorithms together with a samplable secret key space $\mathcal{SK}$, and spaces of public keys $\mathcal{PK}$ and ciphertexts $\mathcal{C}$. As for kuKEM, algorithm up takes an associated-data string together with either a public key or a secret key and produces a new public key or secret key, respectively. A shortcut notation for kuPKE algorithms is:

$$\begin{array}{rcccl} \mathcal{SK} & \to & \text{gen} & \to & \mathcal{PK} \\ \mathcal{PK} \times \mathcal{AD} & \to & \text{up} & \to & \mathcal{PK} \\ \mathcal{SK} \times \mathcal{AD} & \to & \text{up} & \to & \mathcal{SK} \end{array} \qquad \begin{array}{rcccl} \mathcal{PK} \times \mathcal{M} & \to & \text{enc} & \to_{\$} & \mathcal{C} \\ \mathcal{SK} \times \mathcal{C} & \to & \text{dec} & \to & \mathcal{M} \cup \{\perp\} \end{array}$$

A kuPKE scheme UE is correct if for synchronously updated public key and secret key, the latter can decrypt ciphertexts produced with the former: $\Pr[\forall n \in \mathbb{N}\ \text{dec}(sk_n, \text{enc}(pk_n, m)) = m : sk_0 \leftarrow_{\$} \mathcal{SK}, pk_0 = \text{gen}(sk_0), \forall i \in [n]\ ad_i \leftarrow_{\$} \mathcal{AD}, pk_{i+1} = \text{up}(pk_i, ad_i), sk_{i+1} = \text{up}(sk_i, ad_i), m \leftarrow_{\$} \mathcal{M}] = 1$.

A secure kuPKE scheme intuitively guarantees that a message, encrypted to public key pk' that was derived from another public key pk

via sequential updates under associated-data from vector $ad \in \mathcal{AD}^*$, cannot be decrypted by a (computationally bounded, or symbolic) adversary even with access to any secret keys, derived via updates from pk's secret key sk under an associated-data vector $ad' \in \mathcal{AD}^*$ such that ad' is not a prefix of ad. Note that this intuitive security notion matches security of HIBE when associated data is being parsed as identity strings.

All remaining building blocks that we consider in this chapter (i.e., dual PRF and BE) are jointly introduced in Chapter 2.

5.5 Intuition for Lower Bound

Our lower bound proof intuitively says that every group ratcheting scheme with better communication complexity than this bound is either insecure, or not correct, or cannot be built from the building blocks we consider. In the following, we first list these considered building blocks and argue why the selection of those is indeed justified (and not too restrictive). We then abstractly explain the symbolic security definition of group ratcheting, and finally sketch the steps of our proof that is formally given in Section 5.7.

5.5.1 Symbolic Building Blocks

The selection of primitives which a group ratcheting construction may use to reach minimal communication complexity in our symbolic model is inspired by the work of Micciancio and Panjwani [MP04]. For their lower bound of communication complexity in multi-cast encryption—which can also be understood as group key exchange—, Micciancio and Panjwani allow constructions to use pseudo-random generators, secret sharing, and symmetric encryption. We instead consider 1. *(dual) pseudo-random functions*, 2. *key-updatable public key encryption* (with functionality and symbolic security guarantees at least as strong as those of *hierarchical identity based encryption*), and 3. *broadcast encryption* and thereby significantly extend the power

of available building blocks. As secret sharing appears to be rather irrelevant in our setting—as well as it is irrelevant in their setting—, we neglect it to achieve better clarity in model and proof.

Bulding Blocks in Related Work To support the justification of our selection, we note that all previous constructions of group ratcheting base on equally or less powerful building blocks than we consider here: The ART construction [CCG+18] relies on a combination of dual PRF and Diffie–Hellman (DH) group. The actual properties used from the DH group can also be achieved by using generic public key encryption (PKE)—as demonstrated by its following successors. TreeKEM as proposed in the MLS initiative [ACDT20, BBM+20b] relies on a PRG and a PKE scheme. TreeKEM with extended forward-secrecy [ACDT20] relies on a PRG and an updatable PKE scheme. The syntax of the latter in combination with the respective computational security guarantees can be considered weaker than our according symbolic variant of kuPKE. Tainted TreeKEM [ACC+19a] relies on a PKE scheme in the random oracle model. Optimally secure Tainted TreeKEM [ACJM20] relies on an HIBE scheme in the random oracle model. As noted before, functionality and security guarantees of HIBE are captured in our symbolic notion of kuPKE. The property of the random oracle that allows for mixing multiple input values of which at least one is confidential to derive a confidential random output can be achieved similarly by using (a cascade of) dual PRF invocations.[6]

Only the post-compromise *insecure* merge-mechanism of DH shares from Causal TreeKEM [Wei19] is not captured in our symbolic model. Turning this mechanism post-compromise *secure* results in multi-party NIKE, which we intentionally exclude.

[6] If the constructions in [ACC+19a, ACJM20] would rely on stronger (security) guarantees of the random oracle model, their practicability might be questionable.

Grammar The grammar definition of the considered building blocks bases on five types of symbols: messages M, secret keys SK, symmetric keys K, public keys PK, and random coins R (which is a terminal type). These types and their relation are specified in the lower right corner of Figure 5.2. For simplicity (and in order to strengthen our lower bound result), we consider algorithms gen and enc interoperable for kuPKE and BE.[7]

Derivation of protected values:
a) $m \in \boldsymbol{M} \implies \boldsymbol{M} \vdash m$
b) $\boldsymbol{M} \vdash k \implies \forall ad\ \boldsymbol{M} \vdash \mathrm{prf}(k, ad)$
c) $\boldsymbol{M} \vdash k_1, k_2 \implies \boldsymbol{M} \vdash \mathrm{dprf}(\{k_1, k_2\})$
d) $\boldsymbol{M} \vdash \mathrm{enc}(pk, \boldsymbol{RM}, m), sk :$
$\mathrm{Fit}(pk, \boldsymbol{RM}, sk) \implies \boldsymbol{M} \vdash m$

Derivation of public values:
g) $\boldsymbol{M} \vdash sk \implies \boldsymbol{M} \vdash \mathrm{gen}(sk)$
h) $\boldsymbol{M} \vdash pk \implies \forall ad\ \boldsymbol{M} \vdash \mathrm{up}(pk, ad)$
i) $\boldsymbol{M} \vdash pk, m \implies \forall \boldsymbol{RM}\ \boldsymbol{M} \vdash \mathrm{enc}(pk, \boldsymbol{RM}, m)$

Derivation of secret keys:
e) $\boldsymbol{M} \vdash sk \implies \forall ad\ \boldsymbol{M} \vdash \mathrm{up}(sk, ad)$
f) $\boldsymbol{M} \vdash sk \implies \forall u\ \boldsymbol{M} \vdash \mathrm{reg}(sk, u)$

Grammar rules:
1. $M \mapsto SK|PK|\mathrm{enc}(PK, \mathcal{S}(\mathbb{N}), M)$
2. $SK \mapsto K|\mathrm{up}(SK, M)|\mathrm{reg}(SK, \mathbb{N})$
3. $K \mapsto R|\mathrm{prf}(K, M)|\mathrm{dprf}(\{K, K\})$
4. $PK \mapsto \mathrm{gen}(SK)|\mathrm{up}(PK, M)$

Figure 5.2: Grammar and derivation rules of building blocks in our symbolic model.

Derivation Rules Symbolic security for the building blocks is defined via derivation rules that describe the conditions under which symbols can be derived from sets of (other) symbols. These rules are defined in Figure 5.2 clustered into those with which protected values can be obtained, with which secret keys can be updated or registered, and with which public values can be obtained.

Rules b) and c) describe the security of (dual) PRFs, rules d), e), and g) to i) describe the security and functionality of kuPKE (and HIBE), and rules d), f), g), and i) describe the security and functionality of BE.

[7] As a simplification we use $\mathbb{N}$ to denote the user input symbol of BE, $\mathcal{S}(\cdot)$ to denote an unordered compilation of multiple such symbols, and $\{\cdot, \cdot\}$ to denote an unordered compilation of two key symbols. For kuPKE encryptions the second parameter in our symbolic model can be ignored.

Rule d), describing the conditions under which a ciphertext can be decrypted, uses predicate Fit that validates the compatibility of public key and secret key (and set of removed registered users). Intuitively, a secret key sk is compatible with a public key pk if all updates for obtaining sk correspond to updates for obtaining pk in the same order and under the same associated data with respect to an initial key pair, or if the former was registered under the main secret key of the latter (details are in Section 5.7.1).

5.5.2 Symbolic Group Ratcheting

The syntax of group ratcheting was introduced in Section 5.2. In the following we map this syntax to the grammar definition above, and shortly give an intuition for the correctness and security of group ratcheting in the symbolic model.

Inputs and outputs of group ratcheting algorithms init, snd, and rcv are random coins $\mathcal{R}$, local user states $\mathcal{ST}$, ciphertexts $\mathcal{C}$, and group keys $\mathcal{K}$. In our grammar these random coins are sets of type R symbols, local states and ciphertexts are sets of type M symbols, and group keys are symbols of type K.

According to this grammar, we require from symbolic constructions of group ratcheting for being *correct* that 1. all outputs of a group ratcheting algorithm invocation can be derived from its inputs via the derivation rules defined above and 2. in each round the group keys, computed by all users, are equal. The first condition is necessary to allow for symbolic adversaries. We note that this condition furthermore implies 'inverse derivation guarantees', meaning that symbols can *only* be obtained via our derivation rules. For example, for inputs IN and outputs OUT of an algorithm invocation, output $k' \in$ OUT with $\mathrm{prf}(k, ad) = k'$ is either also element of set IN (i.e., $k' \in$ IN), or k' is encrypted in a ciphertext contained in set IN, or IN $\vdash k$ holds. We make these inverse derivation guarantees explicit in Section 5.7.6.

Security To transfer the computational security experiment from Section 5.2 to the execution of symbolic attackers against group ratch-

eting, only few small changes are necessary: 1. a symbolic adversary $\mathcal{A}$ follows the above defined derivation rules for an unbounded time, 2. the target of $\mathcal{A}$ is not to distinguish *securely* marked real group keys from random ones but to derive such *securely* marked keys from the ciphertexts, sent in each round, and the states, exposed at the end of each round, with these derivation rules.

A group ratcheting scheme is *secure in the symbolic model* if an unbounded adversary cannot derive any of the securely marked group keys from the combination of all rounds' ciphertexts and exposed states via the above defined rules. The fully formal variant of this definition is in Figure 5.7.

5.5.3 Lower Bound

Using this symbolic framework, we formulate a sketched variant of Theorem 7 that expresses the lower bound of communication complexity for secure (and correct) group ratcheting constructions:

> *Let* GR *be a secure and correct group ratcheting scheme. For every round i in a symbolic execution of* GR *with senders $\boldsymbol{U}_{\mathbf{S}}^{i}$ and exposed users $\boldsymbol{U}_{\mathbf{X}}^{i}$, the number of sent symbols is $|C[i]| \geq |\boldsymbol{U}_{\mathbf{S}}^{i}| \cdot (|\boldsymbol{U}_{\mathbf{S}}^{i-1}| - 1)$.*

For our proof, we consider a symbolic adversary that proceeds as follows:

1. In round $i-2$ a set of members $\boldsymbol{U}_{\mathbf{X}}^{i-2} \subseteq [n]$ with $|\boldsymbol{U}_{\mathbf{X}}^{i-2}| > 1$ is exposed.
2. In subsequent round $i-1$ these exposed users send (i.e., $\boldsymbol{U}_{\mathbf{S}}^{i-1} := \boldsymbol{U}_{\mathbf{X}}^{i-2}$).
3. In round i a non-empty set of members $\emptyset \neq \boldsymbol{U}_{\mathbf{S}}^{i} \subseteq [n]$ sends.

Assuming no user was exposed in any round before or after $i-2$, our symbolic security definition requires the group key in round i to be secure (i.e., not derivable from exposed states and sent ciphertexts up to round i). In order to show that each sender in round i must send

at least $|\boldsymbol{U}_{\mathbf{S}}^{i-1}| - 1$ ciphertexts to establish this secure group key, we analyze the effects of exposures in round $i-2$, sending in round $i-1$, and sending in round i in the following paragraphs.

At the end of round $i-2$ any symbol derivable by users in set $\boldsymbol{U}_{\mathbf{X}}^{i-2}$ is also derivable by the adversary. After generating new secret random coins at the beginning of round $i-1$, users in set $\boldsymbol{U}_{\mathbf{S}}^{i-1}$ can derive symbols, that the adversary cannot derive, from these new random coins and public symbols from their (exposed) state. We call such derivable symbols of types SK, K, and R that the adversary cannot derive *useful secrets*. Symbols of these types that are derivable by the adversary are called *useless secrets* (resulting in two complementary sets). Before sending in round $i-1$, new useful secrets of a user $u^* \in \boldsymbol{U}_{\mathbf{S}}^{i-1}$ are only derivable for u^* itself but not for any other user $u \in [n] \setminus \{u^*\}$. This is because the origin of these new useful secrets are the new secret random coins generated at the beginning of round $i-1$ and no communication took place after their generation yet. Hence, at sending in round $i-1$ users in set $\boldsymbol{U}_{\mathbf{S}}^{i-1}$ share no *compatible* useful secrets with other users. Secrets are called *compatible* if they are equal or if they are registered via rule f) under the same (main) secret key.

We formulate three observations: I) For deriving a public key pk from a set of type R symbols it is necessary according to grammar rule 4. and derivation rules g) and h) (with their inverse derivation guarantees) that its secret key sk (or one of its update-ancestors' secret key sk) is derivable from this set as well. II) For deriving a ciphertext c, encrypted to a public key pk, from a set of type R symbols it is necessary according to grammar rule 1. and derivation rule i) (with its inverse derivation guarantees) that this public key pk is derivable from it as well. III) Unifying all random coins generated by all users up to (including) round $i-1$ except those generated by user $u^* \in \boldsymbol{U}_{\mathbf{S}}^{i-1}$ in round $i-1$ forms a set of type R symbols from which all useful secrets at the beginning of round $i-1$ can be derived except those that are new to user u^* at that point. Combining these observations shows that at the beginning of round $i-1$ no user $u \neq u^*$ can derive public keys to useful secrets of user $u^* \in \boldsymbol{U}_{\mathbf{S}}^{i-1}$. This further implies that user u cannot derive ciphertexts encrypted to such public keys.

As a result, the set of symbols sent by one user $u \in \boldsymbol{U}_{\mathbf{S}}^{i-1}$ in round $i-1$ contains no ciphertexts directed to useful secrets derivable by another user $u^* \in \boldsymbol{U}_{\mathbf{S}}^{i-1} \setminus \{u\}$ that would transport useful secrets between such users.

We further observe: According to the inverse derivation guarantees of rule c), both inputs to a dual PRF invocation must be derivable for deriving its output. As this requires a shared useful secret on input for deriving a shared useful secret as output, also a dual PRF establishes no shared (compatible) useful secrets in round $i-1$. All remaining derivation rules either output no secrets, or are *unidimensional*, meaning that they only immediately derive one (useful) secret from another. As a result, also after receiving in round $i-1$ users in set $\boldsymbol{U}_{\mathbf{S}}^{i-1}$ share no compatible useful secrets.

Sampling random coins before sending in round i again produces no shared compatible useful secrets between users that shared none before. Hence, also before receiving in round i, users in set $\boldsymbol{U}_{\mathbf{S}}^{i-1}$ share no compatible useful secrets. We recall that our symbolic correctness and security definition requires for the given adversary that the shared group key derived in round i (after receiving) is a *useful secret*.

For quantifying the number of ciphertexts sent in round i, we define two *key graphs* $\mathcal{G}_i^{\text{before}}$ and $\mathcal{G}_i^{\text{after}}$ that represent useful secrets as nodes and derivations among them as edges. Secret y being derivable from secret x is represented by a directed edge from x to y. Although inspired by the proof technique of Micciancio and Panjwani [MP04], the use of key (derivation) graphs in our proof is entirely new.

Graph $\mathcal{G}_i^{\text{before}}$ includes a node for each useful secret that exists after receiving in round i and an edge for each derivation among them except for derivations possible only due to ciphertexts sent in round i. Graph $\mathcal{G}_i^{\text{after}}$ contains $\mathcal{G}_i^{\text{before}}$ and additionally includes edges for derivations possible due to ciphertexts sent in round i. Thus, the number of additional edges in $\mathcal{G}_i^{\text{after}}$ equals the number of sent ciphertexts in round i. Mapping our derivation rules to edges is highly non-trivial (e.g., each sent ciphertext must appear at most once). All details are in Definition 2 and Figure 5.8 of the proof in Section 5.7.

The fact that users in set $\boldsymbol{U}_{\mathbf{S}}^{i-1}$ share no compatible useful secrets

before receiving in round i finds expression in graph $\mathcal{G}_i^{\text{before}}$ as follows: Every such user $u \in \boldsymbol{U}_{\mathbf{S}}^{i-1}$ is represented by nodes in a set $\mathcal{V}_u^i$ that stand for its useful secret random coins from rounds $i-1$ and i (the latter only if u also sent in round i). For every pair of users $u_1, u_2 \in \boldsymbol{U}_{\mathbf{S}}^{i-1}$ with $u_1 \neq u_2$ there exists no node in graph $\mathcal{G}_i^{\text{before}}$ that is reachable via a path from a node in set $\mathcal{V}_{u_1}^i$ and a path from a node in set $\mathcal{V}_{u_2}^i$ simultaneously (including trivial paths). In contrast, every set $\mathcal{V}_u^i$ with $u \in \boldsymbol{U}_{\mathbf{S}}^{i-1}$ must contain a node from which a path in graph $\mathcal{G}_i^{\text{after}}$ reaches node v^* that represents the group key in round i.

In graph $\mathcal{G}_i^{\text{before}}$ node v^* was reachable via a path from nodes $\mathcal{V}_u^i$ of at most one user $u \in \boldsymbol{U}_{\mathbf{S}}^{i-1}$. Otherwise v^* would have been a compatible useful secret for two users in set $\boldsymbol{U}_{\mathbf{S}}^{i-1}$ before receiving in round i. Consequently, at least one edge per user $u^* \in \boldsymbol{U}_{\mathbf{S}}^{i-1} \setminus \{u\}$ must be included in $\mathcal{G}_i^{\text{after}}$ in addition to those contained in $\mathcal{G}_i^{\text{before}}$. Hence, $\mathcal{G}_i^{\text{after}}$ contains at least $|\boldsymbol{U}_{\mathbf{S}}^{i-1}| - 1$ more edges than $\mathcal{G}_i^{\text{before}}$, implying that at least $|\boldsymbol{U}_{\mathbf{S}}^{i-1}| - 1$ ciphertexts were sent in round i.

We now observe that invocations of algorithm snd in every round are independent of sets $\boldsymbol{U}_{\mathbf{X}}^j$ for all j, and invocations of algorithm snd in round i are independent of set $\boldsymbol{U}_{\mathbf{S}}^i$. As a consequence, every sender $u \in \boldsymbol{U}_{\mathbf{S}}^i$ must send $|\boldsymbol{U}_{\mathbf{S}}^{i-1}| - 1$ ciphertexts, anticipating the worst case that it is the only sender in that round. Therefore, $|\boldsymbol{U}_{\mathbf{S}}^i| \cdot (|\boldsymbol{U}_{\mathbf{S}}^{i-1}| - 1)$ ciphertexts are sent in (every) round i.

Interpretation This lower bound, formally proved in Section 5.7, describes the best case of communication complexity both within our model but partially also with respect to the real-world: it holds against very weak adversaries for significantly reduced functionality requirements of group ratcheting without any form of required forward-secrecy. Lower bounds, induced by forward-secrecy for group key exchange [MP04], may furthermore apply to practical group ratcheting and therefore increase necessary communication complexity thereof.[8]

[8] We observe that if a group-ratcheting-equivalent of the amortized $\log(n)$ lower bound for forward-secure group key exchange by Micciancio and Panjwani [MP04] applies as a factor on our lower bound, then our construction

We note that our result even applies to any two rounds between which no user sent.

Bypassing our lower bound is possible for constructions that exploit the algebraic structure of elements (which is forbidden in symbolic models), base on building blocks that we do not allow here (e.g., multiparty NIKE), or provide weaker security guarantees (e.g., recover from state exposures only with an additional delay in rounds).

For clarity we note that the key graph concept used here is independent of the tree structure of keys within our upper bound construction in Section 5.6.

5.6 Upper Bound of Communication Complexity

In order to overcome the deficiencies of existing protocols, we postpone the refresh of parts of the key material in the group by one operation. The resulting construction closely (up to a factor of $\approx \log(n/t)$) meets our communication complexity lower bound.

For computational security of group ratcheting, games $\mathrm{KIND}^b_{\mathsf{GR}}$ from Figure 5.1 are slightly adapted to additionally require immediate forward-secrecy. We note that the use of (a weak form of) kuPKE instead of standard PKE in our construction is only due to required forward-secrecy. Furthermore we emphasize that this used weak kuPKE can be efficiently built from standard assumptions (see e.g., a construction from DDH in [JMM19]).

5.6.1 Construction

Our construction uses ideas from the complete subtree method of broadcast encryption [NNL01] and resembles concepts from TreeKEM [ACDT20, ACC+19a]. More specifically, the construction bases on a static complete (directed) binary tree structure τ with n leaves (i.e., one leaf per group member), on top of which at every node, there is

from Section 5.6 has optimal communication complexity.

an evolving kuPKE key pair. The secret key at each of the n leaves is known only by the unique user that occupies that leaf. For the remaining nodes we maintain the invariant that the only secret keys in a user's state at a given time are those that are at nodes along the direct path of its corresponding leaf to the root of the tree.

We refer to the children of a node v in a tree as $v.c_0$ (left child) and $v.c_1$ (right child), and its parent as $v.p$. Furthermore we let $i, j, i > j$ be two rounds in which the set of sending group members is non-empty and there is no intermediate round l, $i > l > j$, with non-empty sending set. For simplicity in the description we define $j := i - 1$.

Sending To recover from state exposures, our construction lets senders in round $i - 1$ refresh only their own individual leaf key pair. Senders in round i then refresh all remaining secret keys stored in the local states of round $i-1$ senders (i.e., for nodes on their direct paths to the root) on their behalf. This is illustrated in Figure 5.3. Note that (as explained below in paragraph *Receiving*) all group members collect the senders of round $i-1$ into a set $\boldsymbol{U}_{i-1}$ in the rcv algorithm of round $i - 1$. Our construction, formally defined in Figure 5.4, accordingly lets all senders in a round perform five tasks:

1) To refresh their own individual secret key: Generate a fresh secret key for their corresponding leaf and send the respective public key to the group (lines 11-12, 32).

2) To refresh and rebuild direct paths of last round's senders: Sample a new seed for the leaf of each sender of the last round and encrypt it to the respective sender's (refreshed) leaf public key (lines 15-18). Then derive a seed for each non-leaf node on the direct paths from these leaves to the root using the new seeds at the leaves (line 19). Each seed will be used to deterministically generate a fresh key pair for its node.

3) To share refreshed secrets with members who did not send in the last round: Encrypt the new seed of each refreshed non-leaf node to the public key of its child from which it was not derived

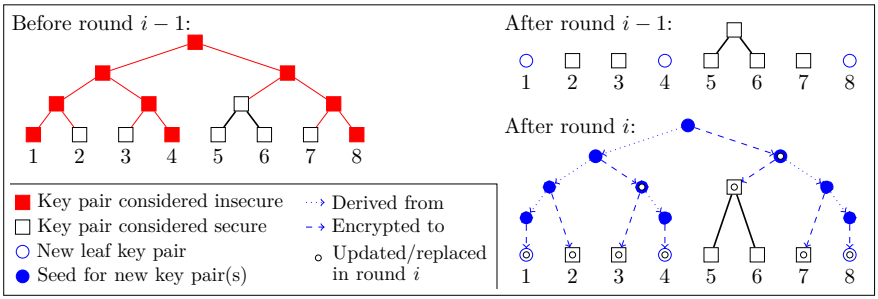

Figure 5.3: Example tree for two rounds $i-1$ and i with $n = 8$, $\boldsymbol{U}_{i-1} = \{1, 4, 8\}$, and $\boldsymbol{U}_i \neq \emptyset$. In round $i-1$, senders generate new key pairs for their leaves. In round i, senders generate seeds for all nodes considered insecure from round $i-1$ and replace leaf key pairs for round $i-1$ senders, as shown in the bottom-right corner.

(lines 21-24, 27-30). Update the used public keys via kuPKE algorithm up (lines 25, 31).

4) To inform the group of changed public keys: Send all changed public keys to the group, including those for which seeds were renewed, and those that were updated via kuPKE (lines 19, 25, 31, 32).

5) Sample and encrypt a group key k for the round to all other users in the group (lines 13, 17, 23, 32).

In step 2), one seed is individually encrypted to each user in set $\boldsymbol{U}_{i-1}$ via public key encryption, which will allow them to reconstruct their direct path in the tree. The purpose of this individual encryption is to let the recent senders forget their old (potentially exposed) secrets and use their fresh secret (which they generated during their last sending) to obtain new, secure secrets on their direct path.

We now describe how all remaining group members are able to rebuild the tree in their view. The reader is invited to follow the explanation and focus their attention on the tree in the lower right corner of Figure 5.3. In this tree, directed edges represent the derivation of a seed at a node from one of its children (dotted) or encryption of a seed

at a node to one of its children (dashed). We consider the Steiner Tree $ST(\boldsymbol{U}_{i-1})$ induced by the set of leaves of users in $\boldsymbol{U}_{i-1}$. $ST(\boldsymbol{U}_{i-1})$ is the minimal subtree of the full tree that connects all of the leaves of $\boldsymbol{U}_{i-1}$ and the root; in the lower right corner tree of Figure 5.3, $ST(\boldsymbol{U}_{i-1})$ is the subtree of blue filled circles and edges between them. For each *degree-one node* v of $ST(\boldsymbol{U}_{i-1})$ (i.e., nodes with only one child in the Steiner Tree), its seed is encrypted to the public key of its child which is not in $ST(\boldsymbol{U}_{i-1})$. This seed can be used to derive some (possibly all) of the secret keys for the nodes on the direct path of v, including v itself (lines 20-25). We denote the set of such degree one nodes of the Streiner Tree as $ST(\boldsymbol{U}_{i-1})_1$ and the child of a node v in $ST(\boldsymbol{U}_{i-1})_1$ that is not in the Steiner Tree as $v.c_{\notin ST(\boldsymbol{U}_{i-1})}$.[9] For each *degree-two node* v of $ST(\boldsymbol{U}_{i-1})$ (i.e., nodes with two children in the Steiner Tree), its seed is encrypted to the public key of its right child (lines 26-31). We denote the set of such degree-two nodes of the Steiner Tree as $ST(\boldsymbol{U}_{i-1})_2$. All of these encrypted seeds are derived from the fresh leaf seeds of users in set $\boldsymbol{U}_{i-1}$ via prf computations, as explained below in paragraph *Construction Subroutines*.

Alongside the seeds, some randomly sampled associated data ad is also encrypted in the ciphertexts of the above paragraph (lines 21, 27). Public keys used for the encryption are afterwards updated with this associated data ad (lines 25, 31). Upon receipt, this associated data is used correspondingly to update the secret keys as well. Due to this mechanism, immediate forward-secrecy is achieved since secret keys stored in users' local states are updated as soon as they are used for decryption.

We refer to the union of nodes that are in the Steiner Tree with nodes that are children of degree-one nodes in the Steiner Tree as $CST = \{v : v \in ST(\boldsymbol{U}_{i-1}) \vee v = w.c_{\notin ST(\boldsymbol{U}_{i-1})} \forall w \in ST(\boldsymbol{U}_{i-1})_1\}$. For step 4) above, senders must publish the new public keys corresponding to all nodes of $CST(\boldsymbol{U}_{i-1})$ (lines 19, 25, 31, 32).

[9] We overload the set theoretic symbol $\notin$ here for brevity.

Proc init(n)
00 $i \leftarrow 1, \boldsymbol{U}_0 \leftarrow \emptyset$
01 $m \leftarrow \mathrm{CBT}(n)$
02 $SK_{\text{init}} \leftarrow_{\$} \mathcal{SK}^m$
03 $PK_\tau \leftarrow \mathrm{genPKTree}(SK_{\text{init}})$
04 $k_{\mathsf{sav}} \leftarrow_{\$} \mathcal{K}$
05 For u from 1 to n:
06 $\quad SK_u \leftarrow \mathrm{getSKPath}(SK_{\text{init}}, u)$
07 $\quad sk^0 \leftarrow \bot$; $sk^1 \leftarrow \bot$
08 $\quad st_u \leftarrow (u, i, PK_\tau, SK_u, \boldsymbol{U}_0, sk^0, sk^1, k_{\mathsf{sav}})$
09 Return $(st_1, \ldots, st_n)$

Proc snd(st)
10 $(u, i, PK_\tau, SK_u, \boldsymbol{U}_{i-1}, sk^0, sk^1, k_{\mathsf{sav}}) \leftarrow st$
11 $sk' \leftarrow_{\$} \mathcal{SK}$
12 $pk' \leftarrow \mathrm{gen}(sk')$
13 $k \leftarrow_{\$} \mathcal{K} \cap \mathcal{M}$
14 $DK[\cdot] \leftarrow \bot$
15 For each $v \in \boldsymbol{U}_{i-1}$:
16 $\quad DK[v] \leftarrow_{\$} \mathcal{K} \cap \mathcal{M}$
17 $\quad ct \leftarrow_{\$} \mathrm{enc}(PK_\tau[v], DK[v]||k)$
18 $\quad CT[v] \leftarrow ct$
19 $(DK_{ST(\boldsymbol{U}_{i-1})}, PK_{ST(\boldsymbol{U}_{i-1})}) \leftarrow \mathrm{genSTree}(DK, \boldsymbol{U}_{i-1})$
20 For each $v \in ST(\boldsymbol{U}_{i-1})_1$:
21 $\quad ad \leftarrow_{\$} \mathcal{AD} \cap \mathcal{M}$
22 $\quad pk \leftarrow PK_\tau[v.c_{\notin ST(\boldsymbol{U}_{i-1})}]$
23 $\quad ct \leftarrow_{\$} \mathrm{enc}(pk, DK_{ST(\boldsymbol{U}_{i-1})}[v]||ad||k)$
24 $\quad CT[v] \leftarrow ct$
25 $\quad PK_{ST(\boldsymbol{U}_{i-1})}[v.c_{\notin ST(\boldsymbol{U}_{i-1})}] \leftarrow \mathrm{up}(pk, ad)$
26 For each $v \in ST(\boldsymbol{U}_{i-1})_2$:
27 $\quad ad \leftarrow_{\$} \mathcal{AD} \cap \mathcal{M}$
28 $\quad pk \leftarrow PK_{ST(\boldsymbol{U}_{i-1})}[v.c_1]$
29 $\quad ct \leftarrow_{\$} \mathrm{enc}(pk, DK_{ST(\boldsymbol{U}_{i-1})}[v]||ad)$
30 $\quad CT[v] \leftarrow ct$
31 $\quad PK_{ST(\boldsymbol{U}_{i-1})}[v.c_1] \leftarrow \mathrm{up}(pk, ad)$
32 $bc \leftarrow (u, pk', CT, PK_{ST(\boldsymbol{U}_{i-1})})$
33 $st \leftarrow (u, i, PK_\tau, SK_u, \boldsymbol{U}_{i-1}, sk^0, sk', k_{\mathsf{sav}})$
34 Return (st, bc)

Proc rcv($st, \boldsymbol{BC}$)
35 $(u, i, PK_\tau, SK_u, \boldsymbol{U}_{i-1}, sk^0, sk^1, k_{\mathsf{sav}}) \leftarrow st$
36 If $\boldsymbol{BC} = \emptyset$:
37 $\quad \boldsymbol{U}_i \leftarrow \boldsymbol{U}_{i-1}$
38 $\quad$ skip to line 56
39 $\boldsymbol{U}_i \leftarrow \emptyset$
40 Let $bc^* \in \boldsymbol{BC}$ be first in some definite order
41 $(v, pk', CT, PK_{ST(\boldsymbol{U}_{i-1})}) \leftarrow bc^*$
42 If $u \in \boldsymbol{U}_{i-1}$:
43 $\quad k_{\mathsf{der}}||k \leftarrow \mathrm{dec}(sk^0, CT[u])$
44 $\quad v^* \leftarrow u$
45 Else:
46 $\quad v^* \leftarrow \mathrm{getSNode}(u, ST(\boldsymbol{U}_{i-1}))$
47 $\quad sk \leftarrow SK_u[v^*.c_{\notin ST(\boldsymbol{U}_{i-1})}]$
48 $\quad k_{\mathsf{der}}||ad||k \leftarrow \mathrm{dec}(sk, CT[v^*])$
49 $\quad SK_u[v^*.c_{\notin ST(\boldsymbol{U}_{i-1})}] \leftarrow \mathrm{up}(sk, ad)$
50 $(SK'_u, PK'_\tau) \leftarrow \mathrm{Rebuild}(st, PK_{ST(\boldsymbol{U}_{i-1})}, CT, k_{\mathsf{der}}, v^*)$
51 For all $bc \in \boldsymbol{BC}$:
52 $\quad (v, pk', CT, PK_{ST(\boldsymbol{U}_{i-1})}) \leftarrow bc$
53 $\quad \boldsymbol{U}_i \leftarrow \boldsymbol{U}_i \cup \{v\}$
54 $\quad PK'_\tau[v] \leftarrow pk'$
55 $k_{\mathsf{sav}} \leftarrow k$
56 $k_{\mathsf{out}} \leftarrow \mathrm{prf}(k_{\mathsf{sav}}, \mathtt{out})$
57 $k_{\mathsf{sav}} \leftarrow \mathrm{prf}(k_{\mathsf{sav}}, \mathtt{sav})$
58 $sk^0 \leftarrow sk^1$
59 $i' \leftarrow i + 1$
60 $st \leftarrow (u, i', PK'_\tau, SK'_u, \boldsymbol{U}_i, sk^0, sk^1, k_{\mathsf{sav}})$
61 Return (st, k_{out})

Figure 5.4: Construction of concurrent group ratcheting in the computational model. $\mathrm{CBT}(n)$ calculates the number of nodes in a complete binary tree with n leaves. $\mathrm{getSNode}(u, ST(\boldsymbol{U}_{i-1}))$ finds the first node v on the direct path of u that is in $ST(\boldsymbol{U}_{i-1})$.

Receiving For rounds in which no member sent, the recipients forward-securely derive symmetric keys (one output group key, and one saved key) from last round's secrets (lines 56-57). In addition, they assign $\boldsymbol{U}_i \leftarrow \boldsymbol{U}_{i-1}$ (line 37), so that senders of subsequent rounds can refresh the secrets of the senders of round $i-1$.

In case members sent in a round, a receiver determines the first message bc^* among all sent in this round, via some definite order (e.g., lexicographic). The receiver then retrieves from this message the ciphertext set CT for decrypting the symmetric secret k and the first seed needed to rebuild the tree: If the receiver sent in the last active round (in which anyone sent), it uses its individual (fresh) secret key (lines 43-44). Otherwise, it uses the secret key of the first node on its direct path that is the child of some node in $ST(\boldsymbol{U}_{i-1})$ (lines 45-48). The decrypted seed, as well as the rest of CT, and the public keys of the Steiner Tree within bc^* are then used to rebuild the secret path for the receiver, as well as the public key tree, as described below in paragraph *Construction Subroutines* (line 50). The resulting symmetric secret is then used to derive the output group key and a new saved key (as described above for rounds without ciphertexts).

Additionally, secret keys used to decrypt ciphertexts (including those as described in the *Construction Subroutines* paragraph below), are updated with the associated data that was also decrypted from the respective ciphertexts (lines 48, 49, 80, 81). Finally, all senders of the round are collected into $\boldsymbol{U}_i$ and their new public keys are saved (lines 51-54) in order to (later) achieve post-compromise security.

Construction Subroutines In the common initialization algorithm init, a complete binary tree of n leaves with a public key at each node is initialized using a list of corresponding secret keys SK_{init} with procedure $PK_\tau \leftarrow \text{genPKTree}(SK_{\text{init}})$ (line 03). Also, the secret keys along the direct path to the root of leaf u for each user are retrieved for that user, using $SK_u \leftarrow \text{getSKPath}(SK_{\text{init}}, u)$.

Figure 5.5 details the subroutines for genSTree and Rebuild (lines 19 and 50). Subroutine genSTree is used in the snd algorithm to compute

the seeds and public keys at each node of the Steiner tree $ST(\boldsymbol{U}_{i-1})$ using the seeds $DK[v]$ sampled for the leaves $v \in \boldsymbol{U}_{i-1}$ (lines 15-18). For each $v \in \boldsymbol{U}_{i-1}$, the receiver uses $DK[v]$ to compute the node's secret key, public key, and (possibly) the seed to be used for its parent (lines 66-69), continuing up the tree until there has already been a seed generated for some node w on the path.

Rebuild is used in the rcv algorithm, by each user u to rebuild its 'secret key path' as well as the 'public key tree' using the public keys of the Steiner Tree $PK_{ST(\boldsymbol{U}_{i-1})}$, the set of ciphertexts CT, and the seed k_{der} obtained from CT corresponding to a node v^* in the tree. First, for every $v \in CST(\boldsymbol{U}_{i-1})$, the receiver sets its public key to that which is in the dictionary $PK_{ST(\boldsymbol{U}_{i-1})}$ (lines 73-74). Then, starting from node v^* using k_{der}, the receiver derives the secret key for v^* and a new seed for its parent if the node is the left child of its parent. Otherwise the receiver uses the secret key just derived to decrypt the seed to be used at its parent (lines 76-82). The receiver continues up the tree until the root is reached.

Proc genSTree$(DK, \boldsymbol{U}_{i-1})$
62 $DK_{ST(\boldsymbol{U}_{i-1})}[\cdot] \leftarrow \perp$; $PK_{ST(\boldsymbol{U}_{i-1})}[\cdot] \leftarrow \perp$
63 For each $v \in \boldsymbol{U}_{i-1}$ from left to right:
64 $\quad k_{\mathsf{der}} \leftarrow DK[v]$
65 $\quad$ While $DK_{ST(\boldsymbol{U}_{i-1})}[v] = \perp$ and $v \neq r$:
66 $\quad\quad DK_{ST(\boldsymbol{U}_{i-1})}[v] \leftarrow k_{\mathsf{der}}$
67 $\quad\quad k'_{\mathsf{der}} || sk^v \leftarrow \mathrm{prf}(k_{\mathsf{der}}, \mathtt{der})$
68 $\quad\quad PK_{ST(\boldsymbol{U}_{i-1})}[v] \leftarrow \mathrm{gen}(sk^v)$
69 $\quad\quad v \leftarrow v.p, k_{\mathsf{der}} \leftarrow k'_{\mathsf{der}}$
70 Return $(DK_{ST(\boldsymbol{U}_{i-1})}, PK_{ST(\boldsymbol{U}_{i-1})})$

Proc Rebuild$(st, PK_{ST(\boldsymbol{U}_{i-1})}, CT, k_{\mathsf{der}}, v^*)$
71 $(u, i, PK_\tau, SK_u, \boldsymbol{U}_{i-1}, sk^0, sk^1, k_{\mathsf{sav}}) \leftarrow st$
72 $PK'_\tau \leftarrow PK_\tau$; $SK'_u \leftarrow SK_u$
73 For each $v \in CST(\boldsymbol{U}_{i-1})$:
74 $\quad PK_\tau[v]' \leftarrow PK_{ST(\boldsymbol{U}_{i-1})}[v]$
75 $v \leftarrow v^*$
76 While $v \neq r$:
77 $\quad k'_{\mathsf{der}} || sk^v \leftarrow \mathrm{prf}(k_{\mathsf{der}}, \mathtt{der})$
78 $\quad SK'_u[v] \leftarrow sk^v$
79 $\quad$ If $\mathtt{deg}(v.p) = 2$ and $v = v.p.c_1$:
80 $\quad\quad k'_{\mathsf{der}} || ad \leftarrow \mathrm{dec}(sk^v, CT[v.p])$
81 $\quad\quad SK'_u[v] \leftarrow \mathrm{up}(sk^v, ad)$
82 $\quad v \leftarrow v.p, k_{\mathsf{der}} \leftarrow k'_{\mathsf{der}}$
83 Return (PK'_τ, SK'_u)

Figure 5.5: Subroutines for construction upper bound. $\mathtt{deg}(v)$ refers to the degree of a node v in a tree, i.e. number of children.

Lemma 1 *For every round $i \in [q]$, the communication costs in an execution $(n, \boldsymbol{U}^0_{\mathbf{X}}, \boldsymbol{U}^1_{\mathbf{S}}, \boldsymbol{U}^1_{\mathbf{X}}, \ldots, \boldsymbol{U}^1_{\mathbf{S}}, \boldsymbol{U}^q_{\mathbf{X}})$ of the group ratcheting pro-*

tocol from figures 5.4 and 5.5 are

$$|\mathrm{C}[i]| = \mathcal{O}\left(|\boldsymbol{U}_{\mathbf{S}}^{i}| \cdot |\boldsymbol{U}_{\mathbf{S}}^{i-1}| \cdot \left(1 + \log\left(\frac{n}{|\boldsymbol{U}_{\mathbf{S}}^{i-1}|}\right)\right)\right).$$

We note that $|\mathrm{C}[i]|$ denotes the number of sent items (i.e., ciphertexts and public keys) per round. Their individual length depends on the respectively deployed kuPKE scheme. (In a setting that defines a *security parameter*, the factor with which the communication costs are multiplied is (asymptotically) constant in this security parameter.)

The proof of Lemma 1 is a combination of arguments from Naor et al. [NNL01], whose construction of broadcast encryption inspired our group ratcheting scheme. It has not been contributed by the author of this thesis. Hence, we refer the interested reader to the article on which this chapter bases [BDR20b, BDR20c].

Theorem 6 (informal) *Assuming secure kuPKE (as proposed in [JMM19, ACDT20]) and a secure PRF, the construction of figures 5.4 and 5.5 is a secure group ratcheting scheme according to the forward-secure variant of game* $\mathrm{KIND}^{b}_{\mathsf{GR}}$ *from Figure 5.1, with security loss at most* $(q_{\mathrm{Round}}+1)\cdot((\lceil\log(n)\rceil+1)\cdot\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{PR}}(\mathcal{B}_{\mathsf{PR}})+\lceil\log(n)\rceil\cdot\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{UE}}(\mathcal{B}_{\mathsf{UE}}))$*, where* n *is the number of group members,* q_{Round} *is the number of executed rounds, and* $\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{PR}}(\mathcal{B}_{\mathsf{PR}})$, $\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{UE}}(\mathcal{B}_{\mathsf{UE}})$ *are upper bounds on the advantage of any adversaries* $\mathcal{B}_{\mathsf{PR}}$, $\mathcal{B}_{\mathsf{UE}}$ *against the security of PRF and kuPKE, respectively.*

The sketched strong security requirements for key-updatable PKE in Section 5.4 are only permitted in our symbolic model to strengthen the impact of our lower bound. For the security of our upper-bound construction, formulated in the informal Theorem 6 above, a significantly weaker variant of kuPKE suffices: it is not necessary to handle a divergence between public and secret key of a key pair (due to instructed updates on different associated data). Additionally, 'forward-secrecy' of updates (i.e., confidentiality of ciphertexts generated before an update with respect to secret key exposures after the update) is

only required to be effective on associated data that the adversary does not know. Due to these relaxations, this variant of kuPKE can be instantiated efficiently from standard assumptions (see e.g., a construction from DDH in [JMM19]).

The formal variant of Theorem 6 as well as its proof have not been contributed by the author of this thesis. Hence, we refer the interested reader to the article on which this chapter bases [BDR20b, BDR20c].

5.7 Lower Bound of Communication Complexity

After giving an intuition of the lower bound in Section 5.5, we here provide all details of the symbolic model and the lower bound proof in it. We therefore shortly revisit the considered build blocks' definitions and make the restrictions of their choice more transparent. Subsequently, we introduce the formal definition of correctness and security for group ratcheting in the symbolic model. We then formulate the theorem for our lower bound and finally prove it in our symbolic model.

5.7.1 Used Building Blocks

The lower bound proof bases on the formulation of derivation rules that express the power of utilizable building blocks. According to these rules it is shown that every secure and correct group ratcheting construction, using these building blocks, cannot perform better in terms of communication complexity than specified in the lower bound. As a consequence, the selection of considered building blocks plays a crucial role for the strength of the statement behind the lower bound. We allow potential constructions to use *(dual) pseudo-random functions*, *public key encryption*, *key-updatable public key encryption* (capturing guarantees of *hierarchical identity based encryption*), and *broadcast encryption*.

As mentioned in Section 5.5, our overall proof approach as well

as the selection of allowed building blocks is inspired by the work of Micciancio and Panjwani [MP04]. As their setting is similar to ours, extending the power of selected building blocks in comparison to their selection partially justifies our approach. Another indication that the considered building blocks do not (overly) restrict group ratcheting constructions is that neither of the known previous constructions [CCG+18, BBM+20b, ACDT20, ACC+19a, ACJM20] bases on stronger primitives than we consider here (symbolically).

We additionally explain why considering these primitives in our proof is reasonable. We permit the use of dual pseudo-random functions as they allow (group ratcheting) constructions to combine multiple input secrets such that only one of them needs to be secure in order to derive a secure output secret. Thereby potentially secure fresh secrets can be mixed with potentially secure old secrets to derive secure new secrets. For explaining the consideration of kuPKE, we refer to our result from Chapter 4 that proves equivalence of key-updatable KEMs (which are related to kuPKE in terms of syntax and security) and optimally secure *two-party ratcheting*. For constructions of (suboptimal) *group-ratcheting* the use of kuPKE should consequently be allowed. With our strengthened notion of kuPKE that also captures HIBE we are in line with the recently and concurrently proposed optimally secure group ratcheting construction [ACJM20] that makes use of HIBE. Finally, since our upper bound construction makes use of broadcast encryption concepts, we also allow the use of this primitive for constructions in our symbolic model.

In the following we comprehensibly reintroduce the grammar and derivation rules for all considered building blocks, already provided at once in Section 5.2, one after another.

We first describe the underlying basic rules within our symbolic model that are independent of the building block primitives that we consider. For all following definitions, those defined here always apply.

Grammar Of the four basic types of symbols messages can be secret keys, secret keys can be symmetric keys, and symmetric keys can be

random coins (the latter being a terminal type). More formally:

$$1.\ M \mapsto SK \qquad 2.\ SK \mapsto K \qquad 2.\ K \mapsto R$$

Derivation Rules Our derivation rules can be read as follows: For a set of symbols $\boldsymbol{M}$, $\boldsymbol{M} \vdash m$ means that m can be derived from the elements in set $\boldsymbol{M}$, which is stated by the following rule:

a) $m \in \boldsymbol{M} \implies \boldsymbol{M} \vdash m$

Pseudo-Random Functions From the syntax definition of pseudo-random functions, keys in set $\mathcal{K}$ are of type K and associated-data inputs in set $\mathcal{AD}$ are of type M. Accordingly, grammar rule 3. is modified to model that symmetric keys can be random coins or the outputs of the (dual) pseudo-random function[10]:

3. $K \mapsto R|\mathrm{prf}(K, M)|\mathrm{dprf}(\{K, K\})$

For (symbolic) security, we define that the output key of a (dual) PRF can be derived from a set if the respective secret inputs can be derived from it as well:

b) $\boldsymbol{M} \vdash k \implies \forall ad\ \boldsymbol{M} \vdash \mathrm{prf}(k, ad)$

c) $\boldsymbol{M} \vdash k_1, k_2 \implies \boldsymbol{M} \vdash \mathrm{dprf}(\{k_1, k_2\})$

Public Key Encryption Before introducing the more complex primitives kuPKE and BE, we begin with normal public key encryption. A public key encryption scheme PE is a triple of algorithms $\mathsf{PE} = (\mathrm{gen}, \mathrm{enc}, \mathrm{dec})$ such that:

- $\mathrm{gen}(sk) \to pk$ where sk is of type SK and for public keys pk type PK is introduced
- $\mathrm{enc}(pk, m) \to_{\$} c$ where m and c are of type M

[10] In order to reduce complexity, we neither explicitly introduce a function that maps two keys to a set of these two keys, nor a special type that depicts the set of two keys.

- $\mathrm{dec}(sk, c) \to m$

In order to include PKE into the symbolic grammar, we provisionally change the first and add a fourth rule to model that 1.' messages can be secret keys, public keys, and ciphertexts (obtained from encryptions of messages), and 4. public keys are outputs of the public key generation of secret keys:

1.' $M \mapsto SK|PK|\mathrm{enc}(PK, M)$

4. $PK \mapsto \mathrm{gen}(SK)$

We furthermore provisionally add derivation rule d') to model that, if a secret key and a ciphertext, encrypted to its public key, can be derived from the set of symbols, then also the message, encrypted in this ciphertext, can be derived from it.

d') $\boldsymbol{M} \vdash \mathrm{enc}(pk, m), sk : pk = \mathrm{gen}(sk) \implies \boldsymbol{M} \vdash m$

This rule will be incrementally generalized due to the consideration of the following primitives.

Key-Updatable Public Key Encryption In the syntax of kuPKE only update algorithm up is added compared to normal PKE. Accordingly, we change the second and fourth grammar rules to model that updated public keys and secret keys can be parsed as public and secret keys respectively:

2. $SK \mapsto K|\mathrm{up}(SK, M)$

4. $PK \mapsto \mathrm{gen}(SK)|\mathrm{up}(PK, M)$

For simplicity, our symbolic grammar and derivation rules treat all algorithms of public key encryption, key-updatable public key encryption, and broadcast encryption with analog syntax uniformly (e.g., public keys of key-updatable public key encryption can be used to broadcast encrypt), which strengthens our results.

To model the decryptability of ciphertexts and the update of secret keys, we adapt rule d') to more general rule d") (and simplify its

notation with helper function Fit in the next paragraph) and add another rule. A message, encrypted to the public key pk_s of a secret key sk_s in ciphertext c can be derived from the set of symbols $\boldsymbol{M}$ if both c and some secret key sk_r can be derived from $\boldsymbol{M}$ and sk_s can be derived from sk_r (e.g., via secret key updates). Furthermore, an updated secret key can be derived from the set of symbols if the respective input secret key can be derived from it:

d") $\exists sk_0, (ad_0, \ldots, ad_{s-1}) \wedge \boldsymbol{M} \vdash \mathrm{enc}(pk_s, m), sk_r : pk_0 = \mathrm{gen}(sk_0),$
$s \geq r \geq 0, \forall i \in [s-1]\ pk_{i+1} = \mathrm{up}(pk_i, ad_i), \forall i \in [r-1]\ sk_{i+1} = \mathrm{up}(sk_i, ad_i)$
$\implies \boldsymbol{M} \vdash m$

e) $\boldsymbol{M} \vdash sk \implies \forall ad\ \boldsymbol{M} \vdash \mathrm{up}(sk, ad)$

We emphasize that the update with respect to associated data resembles HIBE secret key delegation with respect to identity strings. For HIBE encryption, instead of having the identity vector as a parameter, the matching public key can be derived via according updates.

Broadcast Encryption In addition to the algorithms of PKE and kuPKE, broadcast encryption defines a registration algorithm reg with which secret keys can be registered from a (main) secret key. Furthermore, the encryption algorithm is extended by an input parameter that determines the set of excluded users whose registered secret keys should not be able to decrypt the encrypted payload.

In order to easily and comprehensibly adapt the grammar rules, we finalize provisional rule 1.' by adding a set parameter to the encryption—an empty set symbol $\mathcal{S}(\mathbb{N}) = \emptyset$ specifically models PKE and kuPKE ciphertexts, and a non-empty set of integer symbols $\emptyset \neq \mathcal{S}(\mathbb{N})$ models ciphertexts broadcast to the complementary subset of potential recipients. Additionally, rule 2. now includes the registration of secret keys from (main) secret keys:

1. $M \mapsto SK|PK|\mathrm{enc}(PK, \mathcal{S}(\mathbb{N}), M)$
2. $SK \mapsto K|\mathrm{up}(SK, M)|\mathrm{reg}(SK, \mathbb{N})$

In order to simplify notation, we do not introduce another abstract type for describing user identities as input to the encryption or registration algorithm. Instead, we use $\mathbb{N}$ to denote user identities here, dissociating it from its mathematical (non-symbolic) structure.

For modeling decryption of broadcast ciphertexts, we present provisional rule d"') only to simplify all decryption rules under subsumed rule d): If a ciphertext and a secret key can be derived from a set of symbols, and the ciphertext was encrypted to a public key that is compatible with this derived secret key (which is validated by function Fit), then the encrypted message can be derived from the set of symbols as well. In addition to that, for every (main) secret key derivable from the set of symbols, also all secret keys that can be registered with it are derivable:

$$\text{d"')}\quad \exists u \notin \boldsymbol{RM} \wedge \boldsymbol{M} \vdash \mathrm{enc}(mpk, \boldsymbol{RM}, m), sk : mpk = \mathrm{gen}(msk), \\ sk = \mathrm{reg}(msk, u) \implies \boldsymbol{M} \vdash m$$

$$\text{d)}\quad \boldsymbol{M} \vdash \mathrm{enc}(pk, \boldsymbol{RM}, m), sk : \mathrm{Fit}(pk, \boldsymbol{RM}, sk) \implies \boldsymbol{M} \vdash m$$

$$\text{f)}\quad \boldsymbol{M} \vdash sk \implies \forall u\ \boldsymbol{M} \vdash \mathrm{reg}(sk, u)$$

Since we use this predicate multiple times within our proof, we formulate the compatibility of a secret key and a public key with the following predicate and thereby simplify rules d") and d"') to rule d):

$$\begin{aligned}\mathrm{Fit}(pk_s, \boldsymbol{RM}, sk_r) = (&\exists sk'_0, (ad_0, \ldots, ad_{s-1}), u \notin \boldsymbol{RM} : pk_0 = \mathrm{gen}(sk'_0), \\ &s \geq r \geq 0, \forall i \in [s-1]\ pk_{i+1} = \mathrm{up}(pk_i, ad_i), \\ &\forall i \in [r-1]\ sk'_{i+1} = \mathrm{up}(sk'_i, ad_i), \\ &(sk_r = sk'_r \vee (sk_r = \mathrm{reg}(sk'_r, u) \wedge s = r)))\end{aligned}$$

This predicate defines a public key pk_s with a set of integers $\boldsymbol{RM}$ compatible with a secret key sk_r, if public key and secret key originate from some initial secret key sk'_0 such that either sk_r was derived from updates under a prefix of the associated-data vector under which pk_s was derived, or both were updated under the same associated-data vector and sk_r was subsequently registered under an integer that is not in set $\boldsymbol{RM}$.

Derivation of Public Values In addition to the deriving rules that describe how to recover secret values, we add three additional rules that describe how public values can be derived:

g) $\boldsymbol{M} \vdash sk \implies \boldsymbol{M} \vdash \mathrm{gen}(sk)$

h) $\boldsymbol{M} \vdash pk \implies \forall ad\ \boldsymbol{M} \vdash \mathrm{up}(pk, ad)$

i) $\boldsymbol{M} \vdash pk, m \implies \forall \boldsymbol{RM}\ \boldsymbol{M} \vdash \mathrm{enc}(pk, \boldsymbol{RM}, m)$

Overall Definitions We define the set of symbols that can be derived (and recovered) from a set $\boldsymbol{M}$ using $\vdash$ according to our derivation rules as $\mathrm{Der}(\boldsymbol{M})$.

5.7.2 Group Ratcheting

Syntax and its mapping to the symbolic model of group ratcheting are already defined in sections 5.2 and 5.5.2, respectively. We remind the reader that inputs and outputs of group ratcheting algorithms init, snd, and rcv are random coins in the form of sets of type R symbols, local user states and ciphertexts in the form of sets of type M symbols, and group keys in the form of type K symbols.

The context of each element in the local state and in ciphertexts (e.g., sender of ciphertexts, etc.) is assumed to be implicitly known by the processing algorithms (and thus outside our model).

Correctness We define correctness via Figure 5.6, for which we require that $\Pr[\mathrm{FUNC}_{\mathsf{GR}}(n, \boldsymbol{U}_{\mathbf{S}}^1, \ldots, \boldsymbol{U}_{\mathbf{S}}^q) \to 1] = 0$ for all $n, q \in \mathbb{N}^2$ and all $\boldsymbol{U}_{\mathbf{S}}^i \subseteq [n]$ for every $i \in [q]$. Intuitively, this means that after every round i in which all users in set $\boldsymbol{U}_{\mathbf{S}}^i$ are active, the computed keys of all group members in set $[n]$ are equal.

Additionally, we only consider constructions that allow for symbolic adversaries: that is, all outputs of an algorithm invocation must be derivable via Der according to rules a)-i) from its inputs.[11]

[11] Note that this condition excludes constructions that encode data, necessary for the derivation, inside the algorithm specification.

By requiring that the outputs OUT of group ratcheting constructions' algorithms are derivable via Der from their inputs IN, also 'inverse derivation guarantees' are implied: for each symbol $x \in \text{OUT}$ it holds that $x \in \text{IN}$, or x is encrypted in a ciphertext that can be obtained from IN, or that the symbols from which x is directly derivable are derivable from IN as well (e.g., for $x = \text{prf}(k, ad)$ it holds that $\text{IN} \vdash k$). For clarity we make these inverse derivation guarantees explicit in Section 5.7.6.

Game $\text{FUNC}_{\mathsf{GR}}(n, \boldsymbol{U}_{\mathbf{S}}^1, \ldots, \boldsymbol{U}_{\mathbf{S}}^q)$	**Proc** $\text{Round}(\boldsymbol{U})$
00 $r \leftarrow_{\$} \mathcal{R}$; $symb \leftarrow 1$	07 Require $\boldsymbol{U} \subseteq [n]$
01 $(st_1, \ldots, st_n) \leftarrow \text{init}(n; r)$	08 For all $u \in \boldsymbol{U}$:
02 If $\exists u \in [n] : st_u \not\subseteq \text{Der}(\{n\} \cup r)$:	09 $r_u \leftarrow_{\$} \mathcal{R}$
03 Stop with 1	10 $der \leftarrow \text{Der}(st_u \cup r_u)$
04 For i from 1 to q:	11 $(st_u, c_u) \leftarrow \text{snd}(st_u; r_u)$
05 Call $\text{Round}(\boldsymbol{U}_{\mathbf{S}}^i)$	12 If $st_u \cup c_u \not\subseteq der$: $symb \leftarrow 0$
06 Stop with 0	13 $\boldsymbol{c} \leftarrow \bigcup_{u \in \boldsymbol{U}} c_u$
	14 For all $u \in [n]$:
	15 $der \leftarrow \text{Der}(st_u \cup c)$
	16 $(st_u, k_u) \leftarrow \text{rcv}(st_u, \boldsymbol{c})$
	17 If $st_u \cup \{k_u\} \not\subseteq der$: $symb \leftarrow 0$
	18 If $\exists u \in [n] : k_u \neq k_1 \vee symb = 0$
	19 Stop with 1
	20 Return

Figure 5.6: Correctness definition of concurrent group ratcheting. Gray marked lines force the construction to allow for symbolic attackers. Note that we treat random coins, instances' states, and ciphertexts as sets (and not single elements) of types R and M, respectively.

Security In Figure 5.7 we show the execution of a symbolic adversary, representing the symbolic security definition. It lets an attacker choose the active instances (i.e., those that send in a round) and the set of exposed instances per round. Depending on these choices, a computed group key in round i is marked insecure if previously exposed group members did not yet contribute new information (by sending) until round i, or after they contributed but until and including round i no user integrated these new contributions into the computation of a

new common secure group key (by responding) in order to recover from their exposures, or a user was exposed after round i.

Line 26 accordingly declares a key secure 1. if in the current round no instances were exposed (i.e., all exposed instances sent once after their exposure) and 2.a. if either the key in the previous round was already secure or 2.b. if any instance was active in the current round (i.e., after all exposed instances sent at least one instance reacted by sending as well). This reflects that the group recovers from exposures if the exposed instances were active at least once after their exposure (see line 28) and if afterwards anyone was active in the group to integrate the new contribution of the exposed instance for computing a secure group key.[12]

Furthermore, line 35 declares all past keys insecure after an exposure such that no forward-secrecy is required. As mentioned before, we only introduce this restriction to show that our lower bound solely bases on required post-compromise security under concurrent sending in group ratcheting.

A group ratcheting scheme is considered insecure if any securely marked key can be derived from all sent ciphertexts in combination with all exposed states via function Der in the symbolic setting (see line 14).

All random coins, generated during the game, are of terminal type R, and are generated independently such that neither can be derived from the others: $\forall r \in \bigcup_{i\in[q],u\in[n]} \mathrm{R}[i,u]\ r \notin \mathrm{Der}(\bigcup_{i\in[q],u\in[n]} \mathrm{R}[i,u] \setminus \{r\})$.

Definition 1 *A group ratcheting scheme* GR *is symbolically secure and correct if for all* $n, q \in \mathbb{N}$ *and all* $\boldsymbol{U}_{\mathbf{S}}^i \subseteq [n]$ *for every* $i \in [q]^+$ *it holds that* $\Pr[\mathrm{FUNC}_{\mathsf{GR}}(n, \boldsymbol{U}_{\mathbf{S}}^1, \ldots, \boldsymbol{U}_{\mathbf{S}}^q) \to 1] = 0$ *according to Figure 5.6, and if for all* $n, q \in \mathbb{N}$ *and all* $\boldsymbol{U}_{\mathbf{S}}^i \subseteq [n]$, $\boldsymbol{U}_{\mathbf{X}}^j \subseteq [n]$ *for every* $i \in [q]^+, j \in [q]$ *it holds that* $\Pr[\mathrm{SYM}_{\mathsf{GR}}(n, \boldsymbol{U}_{\mathbf{X}}^0, \boldsymbol{U}_{\mathbf{S}}^1, \boldsymbol{U}_{\mathbf{X}}^1, \ldots, \boldsymbol{U}_{\mathbf{S}}^q, \boldsymbol{U}_{\mathbf{X}}^q)$

[12] We note that this is not optimally secure in case only one instance is exposed: Then this single exposed instance could simultaneously contribute new information and compute a new secure common group key. For the purpose of proving a lower bound on communication complexity, this relaxation strengthens our statement.

Game $\mathrm{SYM}_{\mathsf{GR}}(n, \boldsymbol{U}^0_{\mathbf{X}}, \boldsymbol{U}^1_{\mathbf{S}}, \boldsymbol{U}^1_{\mathbf{X}}, \ldots, \boldsymbol{U}^q_{\mathbf{S}}, \boldsymbol{U}^q_{\mathbf{X}})$

00 $\boldsymbol{XU} \leftarrow \emptyset$; $\boldsymbol{SEC} \leftarrow \emptyset$
01 $\boldsymbol{XST} \leftarrow \emptyset$; $\boldsymbol{C} \leftarrow \emptyset$
02 $\mathrm{XST}[\cdot] \leftarrow \emptyset$; $\mathrm{C}[\cdot] \leftarrow \emptyset$
03 $\mathrm{K}[\cdot] \leftarrow \perp$; $\mathrm{R}[\cdot,\cdot] \leftarrow \emptyset$
04 $\mathrm{ST}_{\mathbf{S}}[\cdot,\cdot] \leftarrow \emptyset$; $\mathrm{ST}_{\mathbf{R}}[\cdot,\cdot] \leftarrow \emptyset$
05 $r \leftarrow_{\$} \mathcal{R}$; $\mathrm{R}[0,\cdot] \leftarrow r$
06 $(st_1, \ldots, st_n) \leftarrow \mathrm{init}(n; r)$
07 For all $u \in [n]$:
08 $\quad \mathrm{ST}_{\mathbf{R}}[0,u] \leftarrow st_u$
09 Call Expose($\boldsymbol{U}^0_{\mathbf{X}}$)
10 For i from 1 to q:
11 $\quad$ Call Round($\boldsymbol{U}^i_{\mathbf{S}}$)
12 $\quad$ Call Expose($\boldsymbol{U}^i_{\mathbf{X}}$)
13 $\boldsymbol{C} \leftarrow \bigcup_{j\in[q]} \mathrm{C}[i]$
14 If $\exists i' \in \boldsymbol{SEC}$: $\mathrm{K}[i'] \in \mathrm{Der}(\boldsymbol{C} \cup \boldsymbol{XST})$:
15 $\quad$ Stop with 1
16 Stop with 0

Proc Round($\boldsymbol{U}$)

17 Require $\boldsymbol{U} \subseteq [n]$
18 For all $u \in \boldsymbol{U}$:
19 $\quad r_u \leftarrow_{\$} \mathcal{R}$; $\mathrm{R}[i,u] \leftarrow r_u$
20 $\quad (st_u, c_u) \leftarrow \mathrm{snd}(st_u; r_u)$
21 $\quad \mathrm{ST}_{\mathbf{S}}[i,u] \leftarrow st_u$
22 $\mathrm{C}[i] \leftarrow \bigcup_{u\in\boldsymbol{U}} c_u$
23 For all $u \in [n]$:
24 $\quad (st_u, k_u) \leftarrow \mathrm{rcv}(st_u, \mathrm{C}[i])$
25 $\quad \mathrm{ST}_{\mathbf{R}}[i,u] \leftarrow st_u$
26 If $\boldsymbol{XU} = \emptyset \wedge (\boldsymbol{U} \neq \emptyset \vee i-1 \in \boldsymbol{SEC})$:
27 $\quad \boldsymbol{SEC} \leftarrow \{i\}$
28 $\boldsymbol{XU} \leftarrow \boldsymbol{XU} \setminus \boldsymbol{U}$
29 $\mathrm{K}[i] \leftarrow k_1$
30 Return

Proc Expose($\boldsymbol{U}$)

31 Require $\boldsymbol{U} \subseteq [n]$
32 $\boldsymbol{XU} \leftarrow \boldsymbol{XU} \cup \boldsymbol{U}$
33 $\mathrm{XST}[i] \leftarrow \bigcup_{u\in\boldsymbol{U}} st_u$
34 $\boldsymbol{XST} \leftarrow \boldsymbol{XST} \cup \mathrm{XST}[i]$
35 $\boldsymbol{SEC} \leftarrow \boldsymbol{SEC} \setminus [i-1]$
36 Return

Figure 5.7: Security definition of concurrent group ratcheting in our symbolic model. Everything marked gray is only included for simplifying the proof terminology but is irrelevant for the security definition.

$\rightarrow 1] = 0$ *according to Figure 5.7.*

Communication Costs Communication costs in round i are $|\mathrm{C}[i]|$. One can further consider communication costs until round i ($\sum_{j\in[i]} |\mathrm{C}[j]|$) and amortized communication costs per round until round i ($\sum_{j\in[i]} |\mathrm{C}[j]|/i$).

Relation to Previous Lower Bound The setting that we are modeling is conceptually similar to the one considered by Micciancio and Panjwani [MP04] for their lower bound analysis of group key exchange. One could therefore hope for similar lower bounds of communication complexity (i.e., $\log(n)$ ciphertexts per operation) for group ratcheting. The crucial difference is, however, that their lower bound bases on *forward-secrecy* requirements of group key exchange whereas we require no form of forward-secrecy in our model and prove the lower

bound based on *post-compromise security* requirements. Communication complexity under combined forward-secrecy and post-compromise security requirements may therefore increase both bounds accordingly. We leave the analysis of this to future work.

5.7.3 Lower Bound

Within the above defined framework we formulate the lower bound of communication complexity.

Theorem 7 (Lower Bound) *Let* GR *be a group ratcheting scheme, secure and correct according to Definition 1. For every round* $i \in [q]$ *the communication costs in an execution* $(n, \boldsymbol{U}^0_{\mathbf{X}}, \boldsymbol{U}^1_{\mathbf{S}}, \boldsymbol{U}^1_{\mathbf{X}}, \ldots, \boldsymbol{U}^q_{\mathbf{S}}, \boldsymbol{U}^q_{\mathbf{X}})$*, according to Figure 5.7, are* $|\mathrm{C}[i]| \geq |\boldsymbol{U}^i_{\mathbf{S}}| \cdot (|\boldsymbol{U}^{i-1}_{\mathbf{S}}| - 1)$.

The proof of Theorem 7 proceeds in four steps:

1. We use exposures in round $i-2$ to show that exposed senders in round $i-1$ have no common useful secrets until the end of round $i-1$.

2. We then show that, in order to compute a common useful secret in the group (i.e., a secure group key) in round i, a *single* sender in round i must send as many ciphertexts as the number of exposed users in round $i-2$ that sent in round $i-1$.

3. In the next step we show that *all* senders in round i must do the same and thereby send equally many ciphertexts.

4. We finally show that the behavior of senders in rounds $i-1$ and i is independent of exposures in round $i-2$. This concludes the proof.

Terminology

Before we formally prove Theorem 7, we introduce and define terms to simplify the notation in our lower bound proof:

- *Communication* up to round i: $\boldsymbol{CO}_i := \mathrm{Der}(\bigcup_{j\in[i]} \mathrm{C}[j])$
- *Secrets* are elements that are of types SK, K, or R.
  Via function $\mathrm{Sec}(\boldsymbol{IN}) \to \boldsymbol{OUT}$ a set of elements $\boldsymbol{IN}$ of arbitrary type is reduced to the set $\boldsymbol{OUT}$ of those elements that are of the aforementioned types (i.e., $\boldsymbol{OUT} \subseteq \boldsymbol{IN}$ such that exactly those elements $x \in \boldsymbol{IN}$ of type SK, K, or R are in set $\boldsymbol{OUT}$).
- *Useless secrets* represent all the knowledge on secrets that an attacker can gain up to a certain point during the protocol execution from sent ciphertexts and exposed user states. *Useful secrets* represent secrets that can be derived from ciphertexts sent and random coins generated up to a certain point during the protocol execution that are not *useless*. Below we specify three types of useless and useful secrets with respect to time slots during the protocol execution within rounds (before and after sending, and after exposures in a round).
- *Useless secrets before sending* in round i:
  $\overline{\boldsymbol{US}_{\mathbf{S}}}[i] := \mathrm{Sec}(\mathrm{Der}(\bigcup_{j\in[i-1]} \mathrm{XST}[j] \cup \boldsymbol{CO}_{i-1}))$.
- *Useless secrets after sending* in round i:
  $\overline{\boldsymbol{US}_{\mathbf{R}}}[i] := \mathrm{Sec}(\mathrm{Der}(\bigcup_{j\in[i-1]} \mathrm{XST}[j] \cup \boldsymbol{CO}_{i}))$.
- *Useless secrets after exposure* in round i: $\overline{\boldsymbol{US}_{\mathbf{X}}}[i] = \overline{\boldsymbol{US}_{\mathbf{S}}}[i+1]$.
- *Useful secrets before sending* in round i:
  $\boldsymbol{US}_{\mathbf{S}}[i] := \mathrm{Sec}(\mathrm{Der}(\bigcup_{j\in[i],u\in[n]} \mathrm{R}[j,u] \cup \boldsymbol{CO}_{i-1})) \setminus \overline{\boldsymbol{US}_{\mathbf{S}}}[i]$.
- *Useful secrets after sending* in round i:
  $\boldsymbol{US}_{\mathbf{R}}[i] := \mathrm{Sec}(\mathrm{Der}(\bigcup_{j\in[i],u\in[n]} \mathrm{R}[j,u] \cup \boldsymbol{CO}_{i})) \setminus \overline{\boldsymbol{US}_{\mathbf{R}}}[i]$.
- *Useful secrets after exposure* in round i:
  $\boldsymbol{US}_{\mathbf{X}}[i] := \boldsymbol{US}_{\mathbf{R}}[i] \setminus \overline{\boldsymbol{US}_{\mathbf{X}}}[i]$.
- *Compatible secrets* are secrets of which either can be derived from the other, or secrets that are registered from the same main secret key. We therefore define the *compatible intersection* operator $\boxtimes$ as follows:
  $A \boxtimes B := \{x \mid x \in A \cap B \vee \exists msk, u, v : (x = \mathrm{reg}(msk, u), y = \mathrm{reg}(msk, v), x \in A \wedge y \in B \vee x \in B \wedge y \in A)\}$

5.7.4 Proof of Lower Bound

For the proof we successively analyze the symbolic derivations ahead of and responsible for obtaining a group key in round i, inducing the lower bound of communication complexity. We begin with derivable symbols at the beginning of round $i-1$.

Round $i-1$ At the beginning of round $i-1$ (before sending) it holds by definition that each user's random coins in round $i-1$ (all all secrets derivable from them) cannot be derived from any other random coins up to that round (including respectively compatible secrets):

$$\forall u^* \in [n]\ \mathrm{Der}\Big(\bigcup_{j\in[i-2],u\in[n]} \mathrm{R}[j,u] \cup \bigcup_{u\in[n]\setminus\{u^*\}} \mathrm{R}[i-1,u]\Big) \\ \owedge \mathrm{Der}(\mathrm{R}[i-1,u^*]) = \emptyset$$

We now formulate Lemma 2 that generically expresses which secrets can and, more importantly, cannot be derived by exposed users. Intuitively it says that, after being exposed and sampling new secret random coins, a user u^* cannot derive useful secrets that are compatible with any other user's useful secrets. More precisely, taking the exposed symbols in a local state of user u^* (represented by subset X^*) that were derived from independent random coins (represented by set X with $X^* \subseteq \mathrm{Der}(X)$) and unifying them with u^*'s newly generated secret random coins (represented by set Y) will not derive useful secrets that are compatible with useful secrets derived from these independent random coins (i.e., $(\mathrm{Der}(X^* \cup Y) \cap \boldsymbol{US}) \owedge (\mathrm{Der}(X) \cap \boldsymbol{US}) = \emptyset$).

Lemma 2 *Let sets* $X, X^*, Y, \boldsymbol{US}$ *exist with* $\mathrm{Der}(X) \owedge \mathrm{Der}(Y) = \emptyset$, $X^* \subseteq \mathrm{Der}(X)$, $\mathrm{Der}(X^*) \cap \boldsymbol{US} = \emptyset$, $\mathrm{Der}(\{x_1, x_2\}) \cap \boldsymbol{US} \neq \emptyset$ *implies* $\{x_1, x_2\} \cap \boldsymbol{US} \neq \emptyset$, *and all elements in sets* X, Y *are of terminal type* R. *Then it holds in our symbolic model that* $(\mathrm{Der}(X) \cap \boldsymbol{US}) \owedge (\mathrm{Der}(X^* \cup Y) \cap \boldsymbol{US}) = \emptyset$.

We prove Lemma 2 in Section 5.7.5.

According to Lemma 2 the following is implied (where Y are the random coins of user u^* in round $i-1$, X is the set of remaining

random coins up to that round, and X^* is the set of useless secrets at the beginning of round $i-1$):

$$\begin{aligned}
&\forall u^* \in [n]\ \mathrm{Der}\left(\bigcup_{j\in[i-2],u\in[n]} \mathrm{R}[j,u] \cup \bigcup_{u\in[n]\setminus\{u^*\}} \mathrm{R}[i-1,u]\right) \\
&\qquad \not\ni \mathrm{Der}(\mathrm{R}[i-1,u^*]) = \emptyset \\
\implies &\forall u^* \in [n]\ \mathrm{Der}\left(\bigcup_{j\in[i-2],u\in[n]} \mathrm{R}[j,u] \cup \bigcup_{u\in[n]\setminus\{u^*\}} \mathrm{R}[i-1,u]\right) \\
&\qquad \cap\ \boldsymbol{US}_{\mathbf{S}}[i-1] \\
&\qquad \not\ni \mathrm{Der}(\mathrm{R}[i-1,u^*] \cup \overline{\boldsymbol{US}_{\mathbf{S}}}[i-1]) \cap \boldsymbol{US}_{\mathbf{S}}[i-1] = \emptyset
\end{aligned}$$

As public keys pk are not of a terminal type, for all $pk \in \mathrm{Der}(\bigcup_{j\in[i-2],u\in[n]} \mathrm{R}[j,u] \cup \bigcup_{u\in[n]\setminus\{u^*\}} \mathrm{R}[i-1,u])$ there must exist a sk in the same set such that $pk \in \mathrm{Der}(sk)$ according to derivation rules g) and h) with their inverse derivation guarantees and grammar rule 4. Since, furthermore, there exist no compatible useful secrets between the random coins of u^* in round $i-1$ (together with useless secrets) and the remaining random coins up to round $i-1$ (as shown above), no pk is derivable from these remaining random coins such that the respective sk is useful and derivable by u^* at the beginning of round $i-1$ (with its current random coins and useless secrets). Otherwise sk would be a shared useful secret:

$$\begin{aligned}
\forall u^* \in [n]\ \nexists pk \in \mathrm{Der}&\left(\bigcup_{j\in[i-2],u\in[n]} \mathrm{R}[j,u] \cup \bigcup_{u\in[n]\setminus\{u^*\}} \mathrm{R}[i-1,u]\right) : \\
&\mathrm{Fit}(pk, \boldsymbol{RM}, sk), sk \in \mathrm{Der}(\mathrm{R}[i-1,u^*] \cup \overline{\boldsymbol{US}_{\mathbf{S}}}[i-1]) \\
&\cap\ \boldsymbol{US}_{\mathbf{S}}[i-1]
\end{aligned}$$

In order to derive a ciphertext from a set of terminal symbols, the public key to which this ciphertext is encrypted must be derivable from that set, too (according to rule i) and its inverse derivation guarantee).

Hence, neither a ciphertext encrypted to a useful secret key of user u^* (derivable from its random coins in round $i-1$ and useless secrets) can be derived by the remaining users (from their random coins up to round $i-1$). This finally induces that no useful secrets can be transmitted from these remaining users to user u^* by encrypting them in round $i-1$:

$$
\begin{aligned}
&\forall u^* \in [n]\ \nexists pk \in \mathrm{Der}\Big(\bigcup_{j\in[i-2],u\in[n]} \mathrm{R}[j,u] \cup \bigcup_{u\in[n]\setminus\{u^*\}} \mathrm{R}[i-1,u]\Big) : \\
&\qquad \mathrm{Fit}(pk, \boldsymbol{RM}, sk), sk \in \mathrm{Der}(\mathrm{R}[i-1,u^*] \cup \overline{\boldsymbol{US}_{\mathbf{S}}}[i-1]) \\
&\qquad \cap\ \boldsymbol{US}_{\mathbf{S}}[i-1] \\
\Longrightarrow\ &\forall u^* \in [n]\ \nexists c \in \mathrm{Der}\Big(\bigcup_{j\in[i-2],u\in[n]} \mathrm{R}[j,u] \cup \bigcup_{u\in[n]\setminus\{u^*\}} \mathrm{R}[i-1,u]\Big) : \\
&\qquad c = \mathrm{enc}(pk, \boldsymbol{RM}, m), \mathrm{Fit}(pk, \boldsymbol{RM}, sk), \\
&\qquad sk \in \mathrm{Der}(\mathrm{R}[i-1,u^*] \cup \overline{\boldsymbol{US}_{\mathbf{S}}}[i-1]) \cap \boldsymbol{US}_{\mathbf{S}}[i-1] \\
\Longrightarrow\ &\forall u^* \in [n]\ \nexists c \in \mathrm{C}[i-1] \setminus c_{u^*} : \\
&\qquad (st_{u^*}, c_{u^*}) \leftarrow \mathrm{snd}(\mathrm{ST}_{\mathbf{R}}[i-2,u^*]; \mathrm{R}[i-1,u^*]), \\
&\qquad c = \mathrm{enc}(pk, \boldsymbol{RM}, m), \mathrm{Fit}(pk, \boldsymbol{RM}, sk), \\
&\qquad sk \in \mathrm{Der}(\mathrm{R}[i-1,u^*] \cup \overline{\boldsymbol{US}_{\mathbf{S}}}[i-1]) \cap \boldsymbol{US}_{\mathbf{S}}[i-1] \\
\Longrightarrow\ &\forall u^* \in [n]\ \nexists k : c = \mathrm{enc}(pk, \boldsymbol{RM}, k), c \in \mathrm{C}[i-1] \setminus c_{u^*}, \\
&\qquad (st_{u^*}, c_{u^*}) \leftarrow \mathrm{snd}(\mathrm{ST}_{\mathbf{R}}[i-2,u^*]; \mathrm{R}[i-1,u^*]), \\
&\qquad \mathrm{Fit}(pk, \boldsymbol{RM}, sk), \\
&\qquad sk \in \mathrm{Der}(\mathrm{R}[i-1,u^*] \cup \overline{\boldsymbol{US}_{\mathbf{S}}}[i-1]) \cap \boldsymbol{US}_{\mathbf{S}}[i-1]
\end{aligned}
$$

Now, since (5.1) users cannot derive compatible useful secrets from their random coins in round $i-1$ together with useless secrets, and (5.2) the ciphertexts in this round are themselves a subset of useless secrets and (5.3) contain no encrypted useful secrets, these users can neither derive compatible useful secrets after receiving the ciphertexts in round $i-1$ (if they only use their current random coins plus useless

secrets for derivation; see Equation 5.4):

$$
\begin{aligned}
&\forall u_1, u_2 \in [n], u_1 \neq u_2 \\
&\quad \mathrm{Der}(\mathrm{R}[i-1, u_1] \cup \overline{\boldsymbol{US}_{\mathbf{S}}}[i-1]) \\
&\quad \not\Subset \mathrm{Der}(\mathrm{R}[i-1, u_2] \cup \overline{\boldsymbol{US}_{\mathbf{S}}}[i-1]) \cap \boldsymbol{US}_{\mathbf{S}}[i-1] = \emptyset && (5.1) \\
&\wedge\ \mathrm{Sec}(\mathrm{Der}(\mathrm{C}[i-1] \cup \overline{\boldsymbol{US}_{\mathbf{S}}}[i-1])) \subseteq \overline{\boldsymbol{US}_{\mathbf{R}}}[i-1] && (5.2) \\
&\wedge\ \nexists k : c = \mathrm{enc}(pk, \boldsymbol{RM}, k), c \in \mathrm{C}[i-1] \setminus c_{u^*}, u^* \in \{u_1, u_2\}, \\
&\quad (st_{u^*}, c_{u^*}) \leftarrow \mathrm{snd}(\mathrm{ST}_{\mathbf{R}}[i-2, u^*]; \mathrm{R}[i-1, u^*]), \mathrm{Fit}(pk, \boldsymbol{RM}, sk), \\
&\quad sk \in \mathrm{Der}(\mathrm{R}[i-1, u^*] \cup \overline{\boldsymbol{US}_{\mathbf{S}}}[i-1]) \cap \boldsymbol{US}_{\mathbf{S}}[i-1] && (5.3) \\
\Longrightarrow\ &\forall u_1, u_2 \in [n], u_1 \neq u_2 \\
&\quad \mathrm{Der}(\mathrm{R}[i-1, u_1] \cup \overline{\boldsymbol{US}_{\mathbf{R}}}[i-1]) \\
&\quad \not\Subset \mathrm{Der}(\mathrm{R}[i-1, u_2] \cup \overline{\boldsymbol{US}_{\mathbf{R}}}[i-1]) \cap \boldsymbol{US}_{\mathbf{R}}[i-1] = \emptyset && (5.4)
\end{aligned}
$$

It must be noted that in order to derive a (common) useful secret via a dual PRF invocation, the two inputs must be known to each user and one input must be a useful secret. Since no compatible useful secrets exist under the aforementioned conditions, and a transmission via ciphertexts makes the transmitted value useless (since ciphertexts themselves are useless and under the above conditions contain no useful secrets), neither of the inputs of a dual PRF can be both a useful secret and derivable from both users' current random coins and useless secrets.

We thereby conclude that at the end of round $i-1$ all users, exposed in round $i-2$, share no compatible useful secrets, independent of who sent in round $i-1$: $\forall u_1, u_2 \in \boldsymbol{U}_{\mathbf{X}}^{i-2}, u_1 \neq u_2\ \mathrm{Der}(\mathrm{ST}_{\mathbf{R}}[i-2, u_1] \cup \mathrm{R}[i-1, u_1] \cup \mathrm{C}[i-1]) \not\Subset \mathrm{Der}(\mathrm{ST}_{\mathbf{R}}[i-2, u_2] \cup \mathrm{R}[i-1, u_2] \cup \mathrm{C}[i-1]) \cap \boldsymbol{US}_{\mathbf{R}}[i-1] = \emptyset$.

Beginning of Round i The state of a user at the end of round $i-1$ is a subset of what can be derived from the union of its state from the previous round, random coins generated in round $i-1$, and ciphertexts received in this round. Consequently, at the end of round $i-1$ the states of two users, exposed in round $i-2$, neither contain compatible useful secrets. The random coins, generated at the beginning

of round i, are independent for all users and consequently neither contribute information for deriving compatible secrets:

$$\begin{aligned}
&\forall u_1, u_2 \in \boldsymbol{U}_{\mathbf{X}}^{i-2},\ u_1 \neq u_2 \\
&\qquad \mathrm{Der}(\mathrm{ST}_{\mathbf{R}}[i-2, u_1] \cup \mathrm{R}[i-1, u_1] \cup \mathrm{C}[i-1]) \\
&\qquad \not\owedge \mathrm{Der}(\mathrm{ST}_{\mathbf{R}}[i-2, u_2] \cup \mathrm{R}[i-1, u_2] \cup \mathrm{C}[i-1]) \\
&\qquad \cap \boldsymbol{US}_{\mathbf{R}}[i-1] = \emptyset \\
\implies &\forall u_1, u_2 \in \boldsymbol{U}_{\mathbf{X}}^{i-2},\ u_1 \neq u_2 \\
&\qquad \mathrm{Der}(\mathrm{ST}_{\mathbf{R}}[i-1, u_1] \cup \mathrm{R}[i, u_1]) \\
&\qquad \not\owedge \mathrm{Der}(\mathrm{ST}_{\mathbf{R}}[i-1, u_2] \cup \mathrm{R}[i, u_2]) \cap \boldsymbol{US}_{\mathbf{S}}[i] = \emptyset
\end{aligned}$$

Graph Interpretation We now represent useful secrets and their derivation in a graph such that each useful secret is a node and the derivation of one useful secret from another is a directed edge between them. By distinguishing between derivation via (dual) PRF invocations, secret key updates, and secret key registrations on the one hand and derivation via decryption on the other hand, we obtain the number of ciphertexts during sending in round i (i.e., the communication complexity). Before formally defining this graph, we give an intuition for its components:

- *Key graph* in round i is a directed graph in which all useful secrets are represented as nodes. These nodes are connected by either *given edges* or *communication edges*, both modeling the ability to derive one secret, represented by the destination node, from (an)other secret(s), represented by the source node.
- *Given edges* model the derivation of secrets via (dual) PRF invocations ($k_2 = \mathrm{prf}(k_1, ad)$ or $k_2 = \mathrm{dprf}(\{k_{1a}, k_{1b}\})$), secret key updates ($sk_2 = \mathrm{up}(sk_1, ad)$), or secret key registration ($sk_2 = \mathrm{reg}(sk_1, u)$) such that the respective output secret (k_2 or sk_2) is represented by the destination node and the input secret(s) (k_1, one of $\{k_{1a}, k_{1b}\}$, or sk_1) is/are represented by the source node. Neither (dual) PRF invocations nor secret key updates require communication over the broadcast for the derivation of a secret (hence *'given' edges*).

The derivation via dual PRF is represented by an edge that has as source node only one of the two (useful) input secrets.[13]

- *Communication edges* model the derivation of secrets due to the ability of decrypting them from the broadcast communication. That means for a ciphertext $c = e_0(\text{enc}(pk_1, \boldsymbol{RM}, e_1(k_2)))$ with $\text{Fit}^*(pk_1, sk_1)$ the useful secret key sk_1 can be used to derive useful secret k_2. The encrypted secret k_2 is thereby represented by the destination node, the decryption (secret) key sk_1 is represented by the source node, and the respective ciphertext c from the communication up to round i (containing the encrypted secret) is represented by the edge itself. In order to ensure that each ciphertext is represented in the graph at most once (such that the graph can be used to measure communication complexity), only the inner most useful encryption (i.e., decryptable under a useful secret and encrypting a useful secret) within potential nested sequences of encryptions (e_0 and e_1) is considered as a communication edge.[14]

  In order to simplify the representation of broadcast encryption ciphertexts in a consistent manner, decryption with useful *registered* secret keys is mapped onto decryption with their (useful) *main* secret keys. Therefore, predicate Fit^* ignores registered secret keys in the declaration of compatibility between public and secret keys.[15]

In Figure 5.8 we illustrate how given edges (left) and communication edges (right) in the key graph relate to the derivation of useful secrets. For communication edges one can see that a ciphertext, containing an encrypted useful *registered* secret key, is represented (and substituted)

[13] Since both input secrets must be known by a user to derive the output secret, it is sufficient to pick one of the two input secrets for representation. As the key graph only includes useful secrets, the represented input secret must be useful.

[14] Note that our graph includes a representation of all ciphertexts up to round i and not only those that are sent during round i.

[15] We note that this only changes the destination node of edges in the graph. The number of (sent) ciphertexts and the derivation of useful secrets is still modeled consistently.

by a ciphertext that encrypts this secret key's *main* secret key instead. Together with the following predicate Fit*, this consistently maps all derivations on (communication) edges in the tree without changing the number of edges or disrupting the derivation of useful secrets.

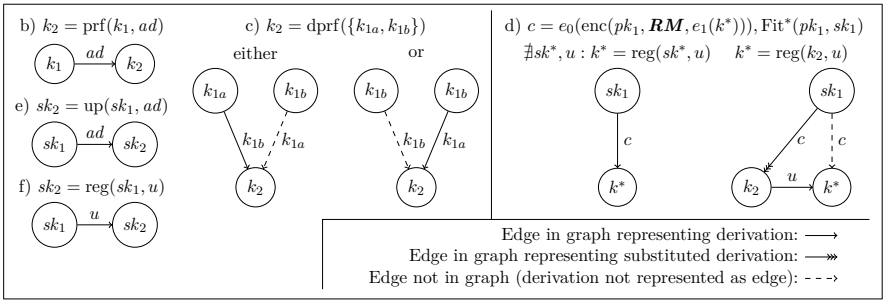

Figure 5.8: Mapping between derivation of useful secrets and their realization in the key graph as given edges (left) and communication edges (right).

The compatibility predicate that neglects registered secret keys is simply defined as follows:

$$\begin{aligned}\mathrm{Fit}^*(pk_s, sk_r) = \big(&\exists sk_0, (ad_0, \ldots, ad_{s-1}) : pk_0 = \mathrm{gen}(sk_0), 0 \geq s \geq r,\\ &\forall i \in [s-1]\ pk_{i+1} = \mathrm{up}(pk_i, ad_i),\\ &\forall i \in [r-1]\ sk'_{i+1} = \mathrm{up}(sk_i, ad_i)\big)\end{aligned}$$

Based on this, we define the key graph:

Definition 2 (Key Graph) *The* key graph *up to round i is a graph $\mathcal{G}_i^{\mathrm{kg}} = (\mathcal{V}_i, \mathcal{E}_i^{\mathrm{giv}} \cup \mathcal{E}_i^{\mathrm{com}})$, where $\mathcal{V}_i = \boldsymbol{US}_{\mathbf{R}}[i]$, the set of* given edges $\mathcal{E}_i^{\mathrm{giv}}$ *contains an edge from one useful secret to another useful secret if the latter can be derived from the former via a (dual) PRF invocation (i.e., $k_2 = \mathrm{prf}(k_1, ad)$ or $k_2 = \mathrm{dprf}(\{k_{1a}, k_{1b}\})$), via a secret key update (i.e., $sk_2 = \mathrm{up}(sk_1, ad)$), or via a secret key registration (i.e., $sk_2 =$*

$\mathrm{reg}(sk_1, u)$*):*

$$\begin{aligned}\mathcal{E}_i^{\mathrm{giv}} =& \{(k_1, k_2) \in \mathcal{V}_i^2 \mid \exists ad : k_2 = \mathrm{prf}(k_1, ad)\} \cup \{(k_{1a}, k_2) \in \mathcal{V}_i^2 \mid \\ & k_2 = \mathrm{dprf}(\{k_{1a}, k_{1b}\}), k_{1b} \in \mathcal{V}_i \cup \overline{\boldsymbol{US}_{\mathbf{R}}[i]}, (k_{1b}, k_2) \notin \mathcal{E}_i^{\mathrm{giv}}\} \\ & \cup \{(sk_1, sk_2) \in \mathcal{V}_i^2 \mid \exists ad : sk_2 = \mathrm{up}(sk_1, ad) \\ & \qquad \vee \exists u : sk_2 = \mathrm{reg}(sk_1, u)\},\end{aligned}$$

and the set of communication edges $\mathcal{E}_i^{\mathrm{com}}$ *contains an edge from one useful secret to another if the latter is, within nested sequences* e_0 *and* e_1 *of encryptions, encrypted under the public key of the former (i.e.,* $c = e_0(\mathrm{enc}(pk_1, \boldsymbol{RM}, (e_1(k_2)))$ *with* $\mathrm{Fit}^*(pk_1, sk_1)$*), unless there exists another encryption to a useful secret within the inner sequence (i.e., there exist* $\mathrm{enc}(pk^*, \boldsymbol{RM}^*, \cdot)$ *with* $\mathrm{Fit}^*(pk^*, sk^*)$ *and* $sk^* \in \mathcal{V}_i$ *within* e_1*):*

$$\begin{aligned}\mathcal{E}_i^{\mathrm{com}} = \{&(sk_1, k_2) \in \mathcal{V}_i^2 \mid c \in \boldsymbol{CO}_i, c = e_0(\mathrm{enc}(pk_1, \boldsymbol{RM}, e_1(k^*))), \\ & \mathrm{Fit}^*(pk_1, sk_1), e_b = e_b^1 \circ \cdots \circ e_b^{l_b}, b \in \{0,1\}, l_b \in \mathbb{N}^0, \\ & e_b^x = \mathrm{enc}(pk^x, \boldsymbol{RM}^x, \cdot), x \in [l_b], \\ & \nexists y \in [l_1] : e_1^y = \mathrm{enc}(pk^y, \boldsymbol{RM}^y, \cdot), \mathrm{Fit}^*(pk^y, sk^y), sk^y \in \mathcal{V}_i, \\ & (k_2 = k^* \wedge \nexists sk^*, u : k^* = \mathrm{reg}(sk^*, u) \vee \exists u : k^* = \mathrm{reg}(k_2, u))\}.\end{aligned}$$

We now let the sets of exposed users in round $i-2$ and senders in round $i-1$ equal (i.e., $\boldsymbol{U}_{\mathbf{X}}^{i-2} = \boldsymbol{U}_{\mathbf{S}}^{i-1}$) such that $|\boldsymbol{U}_{\mathbf{S}}^{i-1}| > 1$ and $|\boldsymbol{U}_{\mathbf{S}}^{i}| > 0$. Furthermore we let no user being exposed in any other round (i.e., $\boldsymbol{U}_{\mathbf{X}}^{j} = \emptyset$ for all $j \neq i-2$). (At the end of the proof we show that no conditions for the sets of exposed users must be enforced in order for the lower bound to hold. We note that the remaining two conditions on the sender set cardinality only exclude the case in which the lower bound collapses to 0.)

As a consequence of these (preliminary) conditions there exists a (cycle-free) path $P_i[u]$ for every user $u \in \boldsymbol{U}_{\mathbf{S}}^{i-1}$ from a node that represents either $\mathrm{R}[i-1, u]$ or $\mathrm{R}[i, u]$ (the latter only if also $u \in \boldsymbol{U}_{\mathbf{S}}^{i}$) to the common group key $\mathrm{K}[i]$ in $\mathcal{G}_i^{\mathrm{kg}}$. This holds because under the above described conditions $\mathrm{K}[i]$ is declared secure (as all exposed

users sent once and obtained a response). Consequently $\mathrm{K}[i]$ must be a useful secret for all secure constructions, derivable for exposed users only through their random coins from the past two rounds (being the only origin of their useful secrets).

From these paths $P_i[u] = (\mathcal{V}_i^{\mathrm{path}}[u], \mathcal{E}_i^{\mathrm{path}}[u])$ we build a (poly)tree $T_i^{\mathrm{path}} = (\bigcup_{u \in \boldsymbol{U}_{\mathbf{S}}^{i-1}} \mathcal{V}_i^{\mathrm{path}}[u], \bigcup_{u \in \boldsymbol{U}_{\mathbf{S}}^{i-1}} \mathcal{E}_i^{\mathrm{path}}[u])$ that models the derivation of the group key (represented as the common leaf) from each exposed user's random coins (represented as individual 'roots' respectively). Without loss of generality, we assume that this (poly)tree is free of cycles.

We now remove all edges e_i^{com} that represent ciphertexts sent in round i from this graph:
$\mathcal{G}_i^{\mathrm{path-com}} = (\bigcup_{u \in \boldsymbol{U}_{\mathbf{S}}^{i-1}} \mathcal{V}_i^{\mathrm{path}}[u], \bigcup_{u \in \boldsymbol{U}_{\mathbf{S}}^{i-1}} \mathcal{E}_i^{\mathrm{path}}[u] \setminus (\mathcal{E}_i^{\mathrm{com}} \setminus \mathcal{E}_{i-1}^{\mathrm{com}}))$.
In this resulting graph $\mathcal{G}_i^{\mathrm{path-com}}$, each exposed user's path is truncated (from the common group key leaf towards each individual root random coins) such that only those useful secrets lay with representatives on their path that were derivable before receiving in round i. In addition to these truncated and thereby disjunct paths, graph $\mathcal{G}_i^{\mathrm{path-com}}$ may contain nodes representing useful secrets (and potentially edges between them that model their derivability) that were generated by other users (who did not send in round $i-1$) and are derivable by the exposed users only though the ciphertexts sent in round i.

From graph $\mathcal{G}_i^{\mathrm{path-com}}$ we extract all weakly connected sub-graphs (i.e., each exposed user's truncated, disjunct path and all remaining weakly connected sub-graphs) and represent each of them as a node in set $\mathcal{V}_i^{\mathrm{node}}$. Nodes in set $\mathcal{V}_i^{\mathrm{node}}$ that represent an exposed user's truncated, disjunct path are called *user-nodes*. These truncated paths, represented by user-nodes, are indeed not connected by a common node since such a common node would otherwise represent a compatible useful secret at the beginning of round i. Consequently, the size of the set of user-nodes $\mathcal{V}_i^{\mathrm{usen}}$ is $|\mathcal{V}_i^{\mathrm{usen}}| = |\boldsymbol{U}_{\mathbf{S}}^{i-1}|$ with $\mathcal{V}_i^{\mathrm{usen}} \subseteq \mathcal{V}_i^{\mathrm{node}}$. We furthermore call the node in set $\mathcal{V}_i^{\mathrm{node}}$ that represents the sub-graph in which the representation of group key $\mathrm{K}[i]$ is contained *key-node* v_i^*. Since the key-node can also be a user-node, the number of user-nodes

that are not the key-node is $|\mathcal{V}_i^{\text{usen}} \setminus \{v_i^*\}| \geq |\boldsymbol{U}_{\mathbf{S}}^{i-1}| - 1$.

We map the source nodes and destination nodes of communication edges added in round i (i.e., sources and destinations of edges in set $(\mathcal{E}_i^{\text{com}} \setminus \mathcal{E}_{i-1}^{\text{com}})$) to their representatives in the set of nodes $\mathcal{V}_i^{\text{node}}$ and unify the resulting edges in set $\mathcal{E}_i^{\text{node}}$. The resulting (poly)tree $T_i^{\text{node}} = (\mathcal{V}_i^{\text{node}}, \mathcal{E}_i^{\text{node}})$ precisely models derivations enabled by ciphertexts sent in round i. That means, each edge in tree T_i^{node} represents a ciphertext sent in round i.

For every node $v \in \mathcal{V}_i^{\text{node}}$ we define its parents as $\text{pa}(v)$, its ancestors (including itself) as $\text{an}(v)$, and the number of all communication edges to it as $\text{ce}(v)$. Formally this means that $\text{an}(v) := \bigcup_{v' \in \text{pa}(v)} \text{an}(v') \cup \{v\}$ and $\text{ce}(v) := |\text{pa}(v)| + \sum_{v' \in \text{pa}(v)} \text{ce}(v')$.

We first consider the case in which the set of sending users in round i only contains one user $\boldsymbol{U}_{\mathbf{S}}^i = \{u^*\}$. We then observe that T_i^{node} is a tree with common leaf v_i^* such that there exist paths to v_i^* from all nodes in $\mathcal{V}_i^{\text{node}}$. Consequently it holds that $\text{an}(v_i^*) = \mathcal{V}_i^{\text{node}}$ implying that $\text{ce}(v_i^*) \geq |\mathcal{V}_i^{\text{node}}| - 1$. Since $\mathcal{V}_i^{\text{usen}}$ is a subset of $\mathcal{V}_i^{\text{node}}$ with $|\mathcal{V}_i^{\text{usen}}| = |\boldsymbol{U}_{\mathbf{S}}^{i-1}|$, the number of ciphertexts user u^* must send in round i for a secure and correct group ratcheting protocol according to Definition 1 is at least $|\boldsymbol{U}_{\mathbf{S}}^{i-1}| - 1$.

Now observe that u^*'s invocation of snd in round i is independent of all sets $\boldsymbol{U}_{\mathbf{X}}^j, j \in [q]$ and set $\boldsymbol{U}_{\mathbf{S}}^i$. Firstly, this implies that user u^* sends as many ciphertexts independent of sets $\boldsymbol{U}_{\mathbf{X}}^j$ for all $j \in [q]$. Secondly, when considering any set of sending users in round i $\boldsymbol{U}_{\mathbf{S}}^i \subseteq [n]$, a correct and secure construction according to Definition 1 must let every user $u \in \boldsymbol{U}_{\mathbf{S}}^i$ independently send as many ciphertexts in round i (as neither of them knows whether also other users send in that round and therefore all of them must anticipate the worst case in which they are the only sender in round i). As a result, the communication complexity in every round i is at least $|\text{C}[i]| \geq |\boldsymbol{U}_{\mathbf{S}}^i| \cdot (|\boldsymbol{U}_{\mathbf{S}}^{i-1}| - 1)$ which proves the lower bound from Theorem 7. ■

Extensions For simplicity and clarity, we only consider here a limited selection of allowed building blocks. The proof, however, shows

that the core issue underlying the lower bound is the inability to mix public cryptographic values into a shared secret non-interactively. Consequently, the list of considered building blocks can be extended manifold without affecting our lower bound.

Even an x-party NIKE for a constant x (e.g., DHKE for $x = 2$) appears to not solve the problem of variable concurrency (entirely): If $t > x$ members concurrently send, subsets of set $[t]$ of size x can compute shared secrets each, but the remaining users cannot derive a corresponding public value for it. Hence, both the remaining senders in the same round and senders in the next round may not be able to utilize this shared secret. We leave the analysis of this as an open question for future work.

5.7.5 Proof of Lemma 2

In order to prove Lemma 2 we (contrarily) assume that two compatible secrets $\{k_1, k_2\} = \{k_1\} \boxtimes \{k_2\}$ (potentially with $k_1 = k_2$) exist with $k_1 \in \mathrm{Der}(X) \cap \boldsymbol{US}$ and $k_2 \in \mathrm{Der}(X^* \cup Y) \cap \boldsymbol{US}$.

For such a tuple (k_1, k_2), there must exist at least one secret $k_0 \in \mathrm{Der}(X) \cap \boldsymbol{US}$ such that $\{k_1, k_2\} \subseteq \mathrm{Der}(\{k_0\})$ because either k_1 can be derived from k_2 (or vice versa and thereby wlog. $k_0 = k_1 = k_2$) or $k_1 = \mathrm{reg}(k_0, u)$ and $k_2 = \mathrm{reg}(k_0, v)$ for some u, v (and thereby $k_1 \notin X$ because all elements in X are of type R but k_1 is not, such that $\{k_1, k_0\} \subset \mathrm{Der}(X)$ which implies that $k_2 \in \mathrm{Der}(X)$). Consequently, it holds that $\{k_1, k_2\} \subset \mathrm{Der}(X) \cap \boldsymbol{US}$.

For secret $k_2 \in \mathrm{Der}(X^* \cup Y) \cap \mathrm{Der}(X) \cap \boldsymbol{US}$ it must hold that $k_2 \notin \mathrm{Der}(X^*) \cup \mathrm{Der}(Y)$ (since $k_2 \in \boldsymbol{US} \implies k_2 \notin \mathrm{Der}(X^*)$ and $k_2 \in \mathrm{Der}(X) \implies k_2 \notin \mathrm{Der}(Y)$). Therefore, in order to fulfill $k_2 \in \mathrm{Der}(X^* \cup Y)$, there must exist some $k^* \in \mathrm{Der}(\{x^*, y\})$ with $k^* \notin \mathrm{Der}(\{x^*\}) \cup \mathrm{Der}(\{y\})$ (and $k_2 \in \mathrm{Der}(\{k^*\})$) for some tuple (x^*, y) such that $x^* \in \mathrm{Der}(X^*), y \in \mathrm{Der}(Y), x^* \notin \mathrm{Der}(Y), y \notin \mathrm{Der}(X^*)$ (since $\mathrm{Der}(X) \boxtimes \mathrm{Der}(Y) = \emptyset \implies \mathrm{Der}(X^*) \cap \mathrm{Der}(Y) = \emptyset$ and $k_2 \notin \mathrm{Der}(X^*) \cup \mathrm{Der}(Y)$). Additionally it must hold that $k^* \in \mathrm{Der}(X) \cap \boldsymbol{US}$ (since $k_2 \in \mathrm{Der}(\{k^*\}) \cap \mathrm{Der}(X) \cap \boldsymbol{US}$, all elements in X are of terminal type R, and according to our symbolic model

for two secrets a, b and set $\boldsymbol{c}$ of terminal type elements it holds that $a \in \mathrm{Der}(\{b\}) \cap \mathrm{Der}(\boldsymbol{c}) \implies b \in \mathrm{Der}(\boldsymbol{c})$). It is important to note that $x^* \notin \boldsymbol{US}$ (as $x^* \in \mathrm{Der}(X^*) \implies x^* \notin \boldsymbol{US}$) and $y \notin \mathrm{Der}(X)$ (as $y \in \mathrm{Der}(Y) \implies y \notin \mathrm{Der}(X)$) must hold.

We summarize: if Lemma 2 does not hold, there must exist a tuple (x^*, y) such that $x^* \in \mathrm{Der}(X^*)$, $x^* \notin \mathrm{Der}(Y) \cup \boldsymbol{US}$, $y \in \mathrm{Der}(Y)$, $y \notin \mathrm{Der}(X^*) \cup \mathrm{Der}(X)$ from which some secret k^* can be derived such that $k^* \in \mathrm{Der}(\{x^*, y\}) \cap \mathrm{Der}(X) \cap \boldsymbol{US}$, $k^* \notin \mathrm{Der}(\{x^*\}) \cup \mathrm{Der}(\{y\})$. The only two derivation rules that combine two elements x^*, y from two sets to a secret k^* are rules c) and d) (note that no rule exist that combines more than two elements and rule i) outputs no secrets).

Rule c) If with respect to rule c) it holds that $k^* = \mathrm{dprf}(\{x^*, y\})$, then in order to fulfill $k^* \in \mathrm{Der}(X)$ (with $y \notin \mathrm{Der}(X)$, meaning that k^* cannot be derived via rule c) from only X) it holds that $k^* \in X$ or for some $\{\tilde{x}_1^1, \tilde{x}_2^1\} \subseteq \mathrm{Der}(X \setminus \{k^*\})$ it holds that $k^* \in \mathrm{Der}^{\mathrm{dec}}(\{\tilde{x}_1^1, \tilde{x}_2^1\})$, where $\mathrm{Der}^{\mathrm{dec}}$ is a derivation under only rule d). In order to substantiate this statement we highlight that only with rule d) secrets can be derived that are also derivable with another rule in our symbolic model (note that by definition of this case, k^* is derivable via rule c) already). If k^* is, according to rule d), derivable via $\tilde{x}_1^1 = \mathrm{enc}(pk, \boldsymbol{RM}, k^*), \mathrm{Fit}(pk, \boldsymbol{RM}, \tilde{x}_2^1)$, then $\tilde{x}_1^1$ (i.e., the ciphertext) must have been derived via rule i) from set $\mathrm{Der}(X) \setminus \{\tilde{x}_1^1\}$ since it is not of a terminal type. This requires again that $k^* \in \mathrm{Der}^{\mathrm{dec}}(\{\tilde{x}_1^2, \tilde{x}_2^2\})$ for some $\{\tilde{x}_1^2, \tilde{x}_2^2\} \subseteq \mathrm{Der}(X \setminus \{k^*\}) \setminus \{\tilde{x}_1^1\}$, or $k^* \in X$ (as the encrypted value k^* must be known in order to apply rule i). Clearly, when all ciphertexts of the form $\tilde{x}_1^j = \mathrm{enc}(pk, \boldsymbol{RM}, k^*)$ with $\mathrm{Fit}(pk, \boldsymbol{RM}, \tilde{x}_2^j)$ are eliminated from set $\mathrm{Der}(X \setminus \{k^*\})$, then there exists no $\{\tilde{x}_1^{l+1}, \tilde{x}_2^{l+1}\} \subseteq \mathrm{Der}(X) \setminus \bigcup_{j \in [l]} \{\tilde{x}_1^j\}$ such that $k^* \in \mathrm{Der}^{\mathrm{dec}}(\{\tilde{x}_1^{l+1}, \tilde{x}_2^{l+1}\})$. Nevertheless, in order to derive any such tuple $(\tilde{x}_1^j, \tilde{x}_2^j)$—which is necessary as these ciphertexts are not of a terminal type—$k^* \in X$ must hold. However, $k^* \in X$ cannot hold as all elements in X are of a terminal type, but k^* can, by the definition of this case, be derived from $\{x^*, y\}$. As a result, k^* is not derivable via rule c).

Rule d) If with respect to rule d) it holds that $x^* = \mathrm{enc}(pk, \boldsymbol{RM}, k^*)$,

$\mathrm{Fit}(pk, \boldsymbol{RM}, y)$ (the proof for the inverse use of variables $y = \mathrm{enc}(pk, \boldsymbol{RM}, k^*)$, $\mathrm{Fit}(pk, \boldsymbol{RM}, x^*)$ is analog), then since pk and x^* are not of terminal types and $X^* \subseteq \mathrm{Der}(X)$ (implying that $\mathrm{Der}(X^*) \subseteq \mathrm{Der}(X)$) it must hold that $x^* \in \mathrm{Der}(X)$. As x^* must be derivable from set $\mathrm{Der}(X)$ via rule i), it must accordingly hold that $pk \in \mathrm{Der}(X)$. This in turn requires either according to rule h) that for the preceding pk^{-1} with $pk = \mathrm{up}(pk^{-1}, ad)$ it holds that $pk^{-1} \in \mathrm{Der}(X)$, or according to rule g) that for the respective sk with $pk = \mathrm{gen}(sk)$ it holds that $sk \in \mathrm{Der}(X)$. If the latter is not the case, then for each preceding pk^{-j} with $pk^{-j} = \mathrm{up}(pk^{-j-1}, ad^{-j})$ it holds that $pk^{-j-1} \in \mathrm{Der}(X)$, or for the respective sk^{-j} with $pk^{-j} = \mathrm{gen}(sk^{-j})$ it holds that $sk^{-j} \in \mathrm{Der}(X)$. As a result, for some sk^{-l} with $\mathrm{Fit}(pk, \boldsymbol{RM}, sk^{-l})$ and $pk \in \mathrm{Der}(sk^{-l})$ it holds that $sk^{-l} \in \mathrm{Der}(X)$. Now, due to $\mathrm{Fit}(pk, \boldsymbol{RM}, sk^{-l})$ with $pk \in \mathrm{Der}(sk^{-l})$ and $\mathrm{Fit}(pk, \boldsymbol{RM}, y)$ it holds that either $sk^{-l} \in \mathrm{Der}(y)$ or $y \in \mathrm{Der}(sk^{-l})$ which contradicts $\mathrm{Der}(X) \cap \mathrm{Der}(Y) = \emptyset$ (because $sk^{-l} \in \mathrm{Der}(X), y \in \mathrm{Der}(Y)$), such that k^* is neither derivable via rule d). ■

5.7.6 Inverse Derivation Guarantees

As part of the enforcement of symbolic algorithm executions, we implicitly require that symbols can *only* be derived *if* their origin can be derived as well, or if they are directly included in the set of symbols. For each of our derivation rules that 'produces' new symbols, we accordingly define an 'inverse' that requires for a derived output that its inputs are derivable, or the output is plain element of the considered set, or that the output is encrypted in a plain element of that set. The latter two alternatives are captured in a separate rule indicated with '$\vdash_{\mathrm{d}}$'.

$\bar{\mathrm{a}})$ $\boldsymbol{M} \vdash m : m \notin \{\mathrm{prf}(k, ad), \mathrm{dprf}(\{k_1, k_2\}), \mathrm{up}(sk, ad), \mathrm{reg}(sk, u), \mathrm{gen}(sk), \mathrm{up}(pk, ad), \mathrm{enc}(pk, \boldsymbol{RM}, m')\} \implies \boldsymbol{M} \vdash_{\mathrm{d}} m$

$\bar{\mathrm{b}})$ $\boldsymbol{M} \vdash \mathrm{prf}(k, ad) \implies \boldsymbol{M} \vdash k \vee \boldsymbol{M} \vdash_{\mathrm{d}} \mathrm{prf}(k, ad)$

$\bar{\mathrm{c}})$ $\boldsymbol{M} \vdash \mathrm{dprf}(\{k_1, k_2\}) \implies \boldsymbol{M} \vdash k_1, k_2 \vee \boldsymbol{M} \vdash_{\mathrm{d}} \mathrm{dprf}(\{k_1, k_2\})$

$$\bar{\text{e}})\ \boldsymbol{M} \vdash \text{up}(sk, ad) \implies \boldsymbol{M} \vdash sk \vee \boldsymbol{M} \vdash_{\text{d}} \text{up}(sk, ad)$$

$$\bar{\text{f}})\ \boldsymbol{M} \vdash \text{reg}(sk, u) \implies \boldsymbol{M} \vdash sk \vee \boldsymbol{M} \vdash_{\text{d}} \text{reg}(sk, u)$$

$$\bar{\text{g}})\ \boldsymbol{M} \vdash \text{gen}(sk) \implies \boldsymbol{M} \vdash sk \vee \boldsymbol{M} \vdash \text{gen}(sk') : \forall \boldsymbol{RM} \text{Fit}(\text{gen}(sk), \boldsymbol{RM}, sk') \vee \boldsymbol{M} \vdash_{\text{d}} \text{gen}(sk)$$

$$\bar{\text{h}})\ \boldsymbol{M} \vdash \text{up}(pk, ad) \implies \boldsymbol{M} \vdash pk \vee \boldsymbol{M} \vdash sk : \forall \boldsymbol{RM}\ \text{Fit}(pk, \boldsymbol{RM}, sk) \vee \boldsymbol{M} \vdash_{\text{d}} \text{up}(pk, ad)$$

$$\bar{\text{i}})\ \boldsymbol{M} \vdash \text{enc}(pk, \boldsymbol{RM}, m) \implies \boldsymbol{M} \vdash pk, m \vee \boldsymbol{M} \vdash_{\text{d}} \text{enc}(pk, \boldsymbol{RM}, m)$$

$$\begin{aligned} \boldsymbol{M} \vdash_{\text{d}} m \implies & m \in \boldsymbol{M} \\ & \vee (\text{enc}(pk, \boldsymbol{RM}, m) \in \boldsymbol{M} \\ & \quad \wedge \boldsymbol{M} \vdash sk : \text{Fit}(pk, \boldsymbol{RM}, sk)) \\ & \vee (\boldsymbol{M} \vdash_{\text{d}} \text{enc}(pk, \boldsymbol{RM}, m) \\ & \quad \wedge \boldsymbol{M} \vdash sk : \text{Fit}(pk, \boldsymbol{RM}, sk)) \end{aligned}$$

5.8 Discussion

We shortly reflect on our construction, compare it to previous works, discuss its limitations with respect to the security model, and propose possible efficiency improvements.

The main purpose of our protocol is to give an upper bound that confirms our lower bound, but not to provide optimal security and maximal functionality under concurrency. Nevertheless, our construction provides the same security as parallel pairwise Signal executions, i.e. FS and PCS one round with non-empty sender set after all exposed users updated their states. In addition, it provides full concurrency for user updates unlike those in [CCG+18, Wei19, BBM+20b, ACDT20, ACC+19a, ACJM20].

We were made aware that our protocol can alternatively be viewed as an adapted (non-trivial and more complex) combination of

1. the propose-then-commit approach from the latest MLS draft [BBM+20b] and
2. Tainted TreeKEM's [ACC+19a] path update for tainted nodes (that allows users to update other users' paths on their behalf).

Indeed, in our protocol, before the senders in $\boldsymbol{U}_i$ perform the snd algorithm, they implicitly *taint* all nodes on paths of senders in $\boldsymbol{U}_{i-1}$, who we view as just having *proposed* updates for their leaves in round $i-1$. (Tainting means that these nodes are marked to be updated by the next active sender(s).) In contrast to the tainting-mechanism in Tainted TreeKEM, in our protocol the nodes on paths of $\boldsymbol{U}_{i-1}$ are immediately *untainted* after round i (i.e., they are not continuously re-updated with every round until one of their descendants updates them).

When using a variant of our construction for dynamic groups, removed members in such groups may maliciously store secrets that they saw during their membership for breaking confidentiality of group secrets after their membership. Effectively solving this problem—discussed as 'double-join'—could be achieved by using ideas from protocols constructed for dynamic groups, such as MLS and Tainted TreeKEM. Without these ideas, it would be required that siblings of all removed users that are still in the group issue state updates before any removed user would be unable to derive the output secrets. Yet, as we discuss below, dynamic member changes appear to happen rather seldom in many practical applications such that this restriction might be insignificant.

Our security model is somewhat weak: we require an honest (but curious) mechanism that clocks rounds, we do not allow the adversary access to random coins used by senders in a round that are not saved to their state, and we do not allow the adversary to alter broadcast messages. Clock synchronization could, however, be rather coarse (resulting in long round periods) as our protocol's speedup in reaching PCS, compared to non-concurrent alternatives that require members to update their states one after another, is already significant. Furthermore, we note that, although our model defines that all members process all ciphertexts in a round, this is not mandatory but allows for

immediate forward-secrecy due to kuPKE key pair updates. Processing all previous ciphertexts before sending, as required for our construction, is usually also unproblematic as sending anyways requires a user to come online, such that all cryptographic operations can be executed at that moment. Especially for reaching authentication and handling out-of-order receipts, tools that are independent of our core state update mechanism can be added, maybe even generically, to our construction. The problem of weak random coins is indeed an open problem for concurrent group ratcheting that we leave for future research.

As stated earlier, it is not ultimately clear whether our lower bound or upper bound is loose (or even both of them). One technique to improve our upper bound would be to utilize more sophisticated broadcast encryption methods than the Complete Subtree method [NNL01], such as the Layered Subset Difference method [HS02] or techniques from the recently proposed optimal broadcast encryption scheme [AY20]. Additionally, if one allows a slight relaxation in the model by allowing for delayed PCS, i.e. PCS in some $\Delta > 1$ rounds, then better communication complexity could be achieved. This is because if users update their state in a given round i by publishing a fresh public key, other users could send secrets to these users to help them recover in all rounds $i' \in \{i+1, i+2, \ldots, i+\Delta\}$, spreading out the communication costs across these rounds and allowing for some adaptivity between senders therein.

5.8.1 Insights for Practice

We shortly summarize concepts from our construction that could enhance, and insights from our lower bound that could influence real-world protocols (like the MLS initiative's design).

Almost-immediate PCS As mentioned many times before, immediate PCS under t-concurrency appears to require t-party NIKE (which is currently inaccessible). Postponing the update of *shared* secrets to a reaction in the next protocol execution step, as implemented

in our construction, bypasses this problem. The major advantages of this bypass are a significant speedup for PCS, compared to sequential state updates, and a maintained balanced tree structure, compared to tree modifications, resulting in a reduced tree depth, or group partitions. An open question remains to analyze our scheme's resilience against weak randomness.

Static Groups are Practical Some deficiencies of our protocol are only relevant in dynamic settings. In contrast, constant groups can benefit from this construction significantly as it maintains communication complexity in all cases nearly optimally. We emphasize that many groups in real-world applications indeed seldom or never change the set of members (e.g., family groups, friendship groups, smaller working groups, etc).

To resolve issues with respect to membership changes, the mechanism proposed in Tainted TreeKEM [ACC+19a] could be applied on path updates in our protocol. Thereby, the 'double-join'-problem could be prevented.

Better Solutions In the light of our lower bound, finding better solutions for reaching PCS under concurrency seems very complicated, if not unlikely. The set of permitted building blocks in our symbolic model is very powerful, the functionality required by constructions in this setting is very restricted, and the adversarial power in the lower bound security definition is very limited. Hence, it seems necessary to utilize more 'exotic' primitives or relax the required PCS guarantees for obtaining better constructions.

6

Systematization of Models for Key Exchange in Groups

Contents

6.1 Introduction 232
6.2 Syntax Definitions 237
6.3 Communication Models 248
6.4 Security Definitions 262
6.5 Discussion 271

Analysis of group key agreement protocols has a long history. Due to the recent advent of ratcheting in groups—as considered in the previous chapter—, interest in these protocols has been renewed. We subsume traditional group key agreement and modern group ratcheting under the term group key exchange (GKE).

Most of the corresponding literature focuses on developing new GKE protocols, so security models only play tangential roles in supporting analysis of particular protocols. In this chapter, we bring the modeling of GKE to the fore by discussing its purposes, taking a fresh look at what GKE tries to achieve within the context of its use, and examining how a model can support that goal. We apply this lens to systematize, classify, and compare characteristics of all relevant GKE models from the literature. From this comparison, we observe a range of shortcomings in existing models, including non generic designs, gaps in coverage of characteristics like overly restrictive syntax notions or unrealistic adversarial capabilities, and incomplete definitions. Our systematization enables us to identify a coherent suite of desirable characteristics—some of them unmet in the literature—that we be-

lieve should be used in GKE models going forward, and we demonstrate that these can be fulfilled by describing a simple and generic model. Given recently expanding interest in secure instant messaging protocols to properly handle the group setting, a clear understanding of security of GKE is of increased importance.

Contributions by the Author The development of our systematization framework and the literature research for this chapter are exclusive contributions by the author of this thesis. Also the textual description in this chapter as well as the design of our new model are primarily contributed by the author. Joint discussions and textual revisions with the remaining authors of the publication in the proceedings of CT-RSA 2021 [PRSS21], on which this chapter bases, crucially improved the results as well as their presentation.

6.1 Introduction

Work on group key exchange (GKE) started with a simple question: *can Diffie–Hellman key exchange be extended to groups of three or more people?* [ITW82, BD95] This simple and general question omitted many aspects of GKE that are important today: dynamic groups (with members continuously joining and leaving the group), groups with offline members (asynchronous mode of operation), or resilience against adversaries with extended access to victims' secrets (ratcheting).

Today, there are many real-world applications that use or could benefit from good group key exchange. These include instant messaging applications as shown in [RMS18] and pursued by IETF's Messaging Layer Security (MLS) initiative [BBM+20a], which aim to provide long-term communication in asynchronous settings, as well as applications such as videoconferencing which are used in synchronous, highly interactive settings.

Many of these demands can be satisfied by existing constructions, but nearly all of the corresponding formal analyses were conducted

in differing models. Consequently, the GKE literature is a zoo of incongruous, heterogeneous security models. Most significantly, there is not even a common core syntax for group key exchange, nor a standard approach for developing GKE security definitions. This has led to a world where almost all models are designed ad hoc, and lack of a common framework or even a common language for discussing GKE which significantly impedes the field's advancement. (Readers of the group key exchange literature will undoubtedly have encountered incompatible terminology and syntax, incomparable models, and even informal or imprecise models and definitions.)

6.1.1 Systemizing Group Key Exchange Models

Given these disparate approaches, we think it is worth taking a step back and having a fresh look at modeling group key exchange, with special attention paid to the environment in which GKE takes place. What features does an application expect of GKE? What type of infrastructure (authentication secrets, network services) does GKE assume exists? What types of adversaries can be considered in the security? A good model should be versatile: it should try to support the requirements of applications in as generic a way as possible, make minimal assumptions on the environment in which it operates, and allow for a wide class of realistic adversaries.

From these principles, our goals are to understand the primitive of group key exchange—its functionality and its demands from users—and identify components of the environment in which GKE is used (e.g., demands from users, interfaces with upper-layer applications, lower-layer network, and adversaries). We aim to derive a taxonomy in which models for GKE can be analyzed, exploring the extent to which models in the literature do or do not meet these demands and restrictions, and the consequences thereof. At the same time, this gives insight into how the field of group key exchange has evolved, and how models in the literature relate to each other. Understanding the history of GKE and having a fresh view on the aims of GKE, we look to the future by examining how these demands and restrictions

can be implemented in a simple, compatible, and versatile manner to support future modeling of GKE.

We organize our investigation around four categories of properties of GKE models:

1. the syntax of GKE (Section 6.2),
2. the definition of partnering (Section 6.3.1),
3. the definition of correctness (Section 6.3.2), and
4. the definition of security (Section 6.4).

While syntax, correctness, and security are present in security definitions throughout cryptography, partnering—a mechanism for determining which computed keys are related to each other, within an environment in which executions are taking place—plays a central role in the key agreement literature (almost) exclusively.

For each of these four categories, we discuss their purposes and central features and classify the literature with respect to them. Having both considered the literature and revisited GKE with a fresh view, we identify desirable characteristics in each of the four categories, from the perspective of generality of use and minimality of assumptions on the context in which GKE takes place. Consequently, we see how individual definitional approaches and, to some extent, subparadigms of group key exchange, do not fully satisfy the needs of group key exchange analysis. We are further able to synthesize a coherent set of desirable properties into a single, generic model, demonstrating it is possible to design a model that simultaneously incorporates these characteristics. Along the way, we hope this chapter gives the reader some new perspectives on at least some aspects of GKE models.

Choice of Literature Group key exchange has a long history of, partially informal, construction-driven literature: publications' contributions mostly consist of proposing new group key exchange schemes, sometimes accompanied with only heuristic security arguments. We

here only consider papers with *formal computational game-based security models*. Our comparison covers all publications on group key exchange with this type of model that appeared in cryptographic 'tier-one' proceedings[1] [BCPQ01, BCP02a, BCP02b, KY03, KLL04, KS05, CCG+18, ACDT20]. Beyond that, we browsed through all relevant 'tier-two' proceedings[2] and selected publications that explicitly claim to enhance the modeling of GKE [GBG09, YKLH18]. We also include three recently published articles on group ratcheting (aka. *continuous* group key exchange), one of which is yet only available as a preprint [CCG+18, ACDT19, ACC+19b].[3] Finally, for purely didactic reasons, we add our rather artificial, restricted, and specialized computational group ratcheting model from Chapter 5.

Tables 6.1, 6.2, 6.5, and 6.3 summarize and compare common features identified from among the models under consideration, along with our identification of desired realizations. The models in these tables are arranged in three clusters: leftmost: group key exchange in static groups [BCPQ01, BCP02b, KY03, KS05, GBG09, CCG+18] including our model from Chapter 5; in the centre: group key exchange with ratcheting [CCG+18, ACDT19, ACC+19b] and our model from Chapter 5; and rightmost: group key exchange in dynamic groups [ACDT19, ACC+19b, BCP01, BCP02a, KLL04, YKLH18]. Within each cluster, models are ordered almost chronologically.

Relation to Two-Party Key Agreement While our focus is on group key exchange, many of the issues here also affect two-party key agreement. In our comparison, we indicate which properties are specific to *group* key exchange, and which apply to key exchange in general. Given the vast two-party key agreement literature, we do not attempt to provide more direct comparisons between two-party and group key exchange.

[1] CRYPTO, Eurocrypt, Asiacrypt, CCS, S&P, and Journal of Cryptology.
[2] TCC, PKC, CT-RSA, ACNS, ESORICS, CANS, ARES, ProvSec, FC etc.
[3] As our analysis was conducted before [ACDT20] was submitted to CRYPTO 2020, we consider a fixed preprint version [ACDT19] here.

Proposed Model Since none of the models we examine achieves all the desirable properties we identify, we finally propose a simple and generic GKE model in the sections 6.2.5, 6.3.4, and 6.4.1 that overcomes the described shortcomings. To be clear: it is *not* our goal to guide the literature to a unified GKE model. Some modeling design decisions are not universal and cannot be reduced to objective criteria, so we are not under the illusion that a perfectly unified model exists, nor that the research community will ever agree upon one notion. Our purpose in writing down a model is to demonstrate the compatibility of the desirable properties.

6.1.2 Basic Notions in Group Key Exchange

A group key exchange scheme is a tuple of algorithms executed by a group of participants with the minimal result that one or multiple (shared) symmetric keys are computed.

Terminology of GKE A **global session** is a joint execution of a GKE protocol. By joint execution we mean the distributed invocation of GKE algorithms by participants that influence each other through communication over a network, eventually computing (joint) keys. Each *local* execution of algorithms by a participant is called a **local instance**. Each local instance computes one or more symmetric keys, referred to as **group keys**. Each group key computed by a single local instance during a global session has a distinct **context**, which may consist of: the set of designated participants, the history of previously computed group keys, the algorithm invocation by which its computation was initiated, etc. Participants of global sessions, represented by their local instances, are called **parties**. (We discuss the as-yet unsettled relation between local instances and parties and their participation in sessions in Section 6.5.1.) If the set of participants in a global session can be modified during the life of the session, this is **dynamic** GKE; otherwise it is **static** GKE.

There are many alternative terms used in the GKE literature for these ideas: local instances are sometimes called *processes*, local *ses-*

sions, or (misleadingly) *oracles*; group keys are sometimes called *session keys*; and parties are sometimes called *users*.

Security Models for GKE As in most game-based models, an adversary against the security of a GKE scheme plays a game with a challenger that simulates multiple parallel real global sessions of the GKE scheme. The challenge that the adversary is (almost always) asked to solve is to distinguish whether a **challenge key** is a real group key established in one of the simulated global sessions or is a random key. In order to solve this challenge, the adversary is allowed to obtain group keys that were computed independently (called **key reveal**), local secrets of instances that do not enable the trivial solution of the challenge (called **state exposure**), and static party secrets that neither trivially invalidate the challenge (called **corruption**).

6.2 Syntax Definitions

Modeling a cryptographic primitive starts with fixing its syntax: the set of algorithms that are available, the inputs they take and the outputs they generate. Some of these algorithms bridge between different layers in a protocol stack, e.g., they take input from a higher level application and produce a ciphertext that is to be transported via a lower level network. As a large number of design choices are possible at each of these layers, and these choices have to be reflected in the primitive's syntax, a single canonic syntax for GKE did not yet crystallize. Indeed, we categorized the GKE models we consider according to the most important classes of syntactical design choices and observe that no two models have identical profiles.[4] Roughly, we distinguished models by (1) imposed limits on the number of supported parties, sessions, and instances; (2) the assumptions that are made on the available infrastructure (e.g., the existence of a PKI); (3) the type of operations that the protocols implement (adding users, removing

[4]Surprisingly, this holds even for models that appeared in close succession in publications of the same authors.

users, refreshing keys, ...); and (4) the information that the protocols provide to the invoking application (set of group members, session identifier, ...). We compiled the results of our studies in Table 6.1. If in any of the categories one option is clearly more attractive than the other options, we indicate this in the **Desirable** column (we leave the cells of that column empty if no clear best option exists). Finally, the **Our model** column indicates the profile of our own GKE model.

The upcoming paragraphs describe our categories in detail. For some models an unambiguous mapping to our categories is not immediate, in which case we made the assignment such that it comes closest to what we believe the authors intended.

Syntax	GKE specific	[BCPQ01]	[BCP02b]	[KY03]	[KS05]	[GBG09]	[CCG+18]	Chapter 5	[ACDT19]	[ACC+19b]	[BCP01]	[BCP02a]	[KLL04]	[YKLH18]	Desirable	Our model
Quantities																
Instances per party	○	*n*	*n*	*n*	*n*	*n*	*n*	1	1	(1)	1	*n*	*n*	*n*	*n*	*n*
Parties per session	●	**F**	**F**	**V**	**V**	**V**	**V**	**V**	**D**	**D**	**D**	**D**	**D**	**D**	**D**	**D**
Multi-participation	●	○	○	○	○	○	○	○	○	○	○	○	○	○	●	●
Setup assumptions																
Authentication by ...	○	**SK**	**PW**	**PK**	**PK**	**PK**	**PK**	-	**PK**	**(PK)**	**SK**	**PK**	**PK**	**PK**		any
PKI	○	-	-	●*	●*	⊙	⊙	-	●*	⊙	-	⊙	●*	●		-
Online administrator	●	-	-	-	-	-	-	●	⊙	◔	●	●	⊙	○	○	○
Operations																
Level of specification	○	○	**G**	○	○	○	**L**	**L**	**L**	**L**	**G**	**G**	**G**	◔	**L**	**L**
Algo: Setup	○	○	●	○	○	○	●	●	●	●	●	●	●	○		-
Algo: Add	●	○	○	○	○	○	○	○	●	●	●	●	●	●		-
Algo: Remove	●	○	○	○	○	○	○	○	●	●	●	●	●	●		-
Algo: Refresh/Ratchet	○	○	○	○	○	○	⊙	●	●	●	○	○	○	○		-
Abstract interface	○	○	○	○	○	○	●	○	○	○	○	○	○	○	●	●
Return values																
Group key	○	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
Ref. for session	●	○	○	⊙	●	⊙	○	○	○	○	○	○	⊙	⊙		○
Ref. for group key	○	○	○	⊙	⊙	○	○	○	○	○	○	○	●	○	●	●
Designated members	●	⊙	⊙	⊙	●	●	⊙	⊙	○	⊙	●	○	⊙	⊙	●	●
Ongoing operation	○	-	⊙	-	-	-	○	○	○	⊙	○	○	○	⊙		○
Status of instance	○	○	○	⊙	○	⊙	⊙	⊙	⊙	⊙	○	○	○	⊙	●	⊙

Table 6.1: Syntax Definitions. Notation: *n*: many; **F**: fixed; **V**: variable; **D**: dynamic; ●: yes; ⊙: implicitly; ◕: almost; ◔: partially; ○: no; -: not applicable; **SK**: symmetric key; **PW**: password; **PK**: public key; **G**: global; **L**: local.

6.2.1 Quantities

All models we consider assume a universe of parties that are potential candidates for participating in GKE sessions. **Instances per party:** While most models assume that each party can participate—using different instances—in an unlimited number of sessions, three models impose a limit to a single instance per party.[5] In Table 6.1 we distinguish these cases with the symbols n and 1, respectively. **Parties per session:** While some models prescribe a fixed number of parties that participate in each GKE session, other models are more flexible and assume either that the number of parties is in principle variable though bound to a static value when a session is created, or even allow that the number of parties changes dynamically over the lifetime of a session (accommodating parties being added/removed). In the table we encode the three cases with the symbols **F**,**V**,**D**, respectively. **Multi-participation:** In principle it could be possible that parties participate multiple times in parallel in the same session (through multiple instances, e.g., from their laptop and smartphone). We note that all of the assessed models exclude such a feature and impose a limit of at most one participation per party, which we encode with symbol ○ in the whole row. We believe however that a multi-participation feature might be useful in certain cases.

Discussion We note that models of type **F** in the Parties-per-session category might be considerably weaker than models of type **V**. For example, security reductions of early ring-based GKE protocols [BD95] require that the number of participants of sessions always be even [BD05] —a restriction that is not desirable in practice yet may not be clearly visible in type **F** models.

[5] The case of [ACC+19b] is somewhat special: While their syntax in principle allows that parties operate multiple instances, their security definition reduces this to strictly one instance per party. For their application (secure instant messaging) this is not a limitation as parties are short-lived and created ad-hoc to participate in only a single session.

6.2.2 Setup Assumptions

Security models are formulated with respect to a set of properties that are assumed to hold for the environment in which the modeled primitive is operated. We consider three classes of such assumptions, related to the pre-distribution of key material to be used for authentication, the availability of a centralized party that leads the group communication, and the type of service that is expected to be provided by the underlying communication infrastructure. **Authentication by ...:** If a GKE protocol provides key establishment with authentication, its syntax has to reflect that the latter is achievable only if some kind of cryptographic setup is established before the protocol session is executed. For instance, depending on the type of authentication, artifacts related to accessing pre-shared secret keys, passwords, or authentic copies of the peers' public keys, will have to emerge. In the table we encode these cases with symbols **SK**,**PW**,**PK**, respectively.[6] **PKI:** In the case of public-key authentication we studied what the models say about how public keys are distributed, in particular whether a public key infrastructure (PKI) is explicitly or implicitly assumed. In the table we indicate this with the symbols ● and ⊙. We further specially mark with ●* the cases of 'closed PKIs' that service exclusively potential protocol participants, i.e., PKIs with which non-participants (e.g. an adversary) cannot register their keys. **Online administrator:** The number of participants in a GKE session can be very large, and, by consequence, properly orchestrating the interactions between them can represent a considerable technical challenge.[7] Two of the models we consider resolve this by requiring that groups be managed by a distinguished always-honest leader who decides which operations happen in which order, further two assume the same but

[6] In continuation of Footnote 5: The case of [ACC+19b] is special in that the requirement is an *ephemeral asymmetric key*, i.e. a public key that is ad-hoc generated and used only once. Also note that our restricted model from Chapter 5 assumes external authentication mechanisms.

[7] Consider, for instance, that situations stemming from participants concurrently performing conflicting operations might have to be resolved, as have to be cases where participants become temporarily unavailable without notice.

without making it explicit, and our model from Chapter 5 requires synchronized clocks for separating rounds. The model of [ACC+19b] is slightly different in that a leader is still required, but it does not have to behave honestly. The model of [YKLH18] does not assume orchestration: Here, protocols proceed execution as long as possible, even if concurrent operations of participants are not compatible with each other. This is argued to be sufficient if security properties ensure that the resulting group keys are sufficiently independent. The remaining models are so simple that they don't require any type of administration.

Discussion While the authentication component that is incorporated into GKE protocols necessarily requires the pre-distribution of some kind of key material, the impact of this component on the GKE model should be minimal; in particular, details of PKI-related operations should not play a role. It should be even less desirable to assume closed PKIs to which outsiders cannot register their keys.

Requiring the existence of an online administrator makes it easy to ensure that all participants in a session have the same view on the communication and group membership list, but also may limit the applicability of the model. For instance, instant messaging protocols are expected to tolerate that participants, including any administrator, might go offline without notice. Also, in certain decentralized environments it wouldn't be clear who would take the role of the leader. On the other hand, if there is no online administrator, yet it shall be ensured that all participants agree on some common decision. For example, regarding the group membership list, it seems necessary to employ involved techniques from distributed computing like a Byzantine Concensus protocol. Note that GKE protocols might choose not to promise common decisions in this respect, but to just communicate locally accurate pictures when delivering keys (see Section 6.2.4).

6.2.3 Operations

In this category we compare the GKE models with respect to the algorithms that parties have available for controlling how they engage in sessions. **Level of specification:** While precisely fixing the APIs of these algorithms seems a necessity for both formalizing security and allowing applications to generically use the protocols, we found that very few models are clear about API details: Four models leave the syntax of the algorithms fully undefined.[8] Another four models describe operations only as global operations, that is, specify how the overall state of sessions shall evolve without being precise about which steps the individual participants shall conduct. Only four models fix a local syntax, that is, specify precisely which participant algorithms exist and which inputs and outputs they take and generate, respectively. In the table, we indicate the three levels of specification with the symbols ○, **G**, and **L**, encoding the terms 'missing', 'global', and 'local', respectively. The model of [YKLH18] sits somewhere between **G** and **L**, and is marked with ◔. **Algo:** The main operations executed by participants are session initialization (either of an empty group or of a predefined set of parties), the addition of participants to a group, the removal of participants from a group, and in some cases a key refresh (which establishes a new key without affecting the set of group members). In the table we indicate which model supports which of these operations. Note that the correlation between the Add/Remove rows and symbol **D** in Quantities/Parties-per-Session is as expected. Only very recent models that emerged in the context of group ratcheting support the key refresh operation. **Abstract interface:** While the above classes Add/Remove/Refresh are the most important operations of GKE, other options are possible, including Merge and Split operations which join two established groups or split them into parts, respectively. In principle each additional algorithm could explicitly appear in the syntax definition of the GKE model, but a downside

[8] In some cases, however, it seems feasible to reverse-engineer some information about an assumed syntax from the security reductions also contained in the corresponding works.

of this would be that the models of any two protocols with slightly different feature sets would become, for purely syntactic reasons, formally incomparable. An alternative is to use only a single algorithm for all group-related operations, which can be directed to perform any supported operation by instructing it with corresponding commands. We believe that this flexible approach towards defining APIs to group operations has quite desirable advantages, but note that only one of the considered models supports it.

Discussion We emphasize once more that we consider the specification of algorithms in an instance-centric fashion, that is, according to the **L**-level of specification, a vital necessity: It is required to achieve both practical implementability and meaningful security definitions. To see the latter, consider that the only way for adversaries to attack (global) sessions is by exposing (local) instances to their attacks.

6.2.4 Return Values

The main outcome of a successful GKE protocol execution is the group key itself. In addition, protocol executions might establish further information that can be relevant for the invoking application. We categorize the GKE models by the type of information conveyed in the protocol outcome. **Group key:** We confirm that all models that we consider have a syntactical mechanism for delivering the key. **Reference for session:** By a session reference we understand a string that serves as an unambiguous handle to a session, i.e., a value that uniquely identifies a distributed execution of the scheme algorithms. Some of the models we consider require that such a string be established as part of the protocol execution, but not necessarily they prescribe that it be communicated to the invoking application along with the key. (Instead the value is used to define key security.) In Table 6.1, we indicate with symbols ● and ⊙ whether the models require the explicit or implicit derivation and communication of a session reference. We mark models with ○ if no such value is considered. **Reference for group key:** A key reference is similar to a session reference but

instead of referring to a session it refers to an established key. While references to sessions and keys are interchangeable in some cases, in general they are not. This is, for instance, trivially the case for protocols that establish multiple keys in a single execution. Further, if communication is not authentic, session references of protocol instances can be matching while key references (and thus keys) are not. In the table we indicate with symbols ● and ⊙ if the models consider explicit or implicit key references. **Designated members:** Once a GKE execution succeeds with establishing a shared key, the corresponding participants should learn who their partners are, meaning, with whom they share the key. In some models this communication step is made explicit, in others, in particular if the set of partners is *input* to the execution, this step is implicit. A third class of models does not communicate the set of group members at all. In the table we indicate the cases with symbols ●, ⊙, ○, respectively. **Ongoing operation:** In GKE sessions, keys are established as a result of various types of actions, particularly including the addition/removal of participants, and the explicit refresh of key material. We document for each considered model whether it communicates for group keys through which operation they were established. **Status of instance:** Instances can assume different protocol-dependent internal states. Common configurations are however that instances can be in an accepted or rejected state, meaning that they consider a protocol execution successful or have given up on it, respectively. In this category we indicate whether the models we consider communicate this status information to the invoking application.

Discussion In settings where parties concurrently execute multiple sessions of the same protocol, explicit references to sessions and/or keys are vital for maintaining clarity about which key belongs to which execution. (Consider attacks where an adversary substitutes all protocol messages of one session with the messages of another session, and vice versa, with the result that the party develops a wrong understanding of the context in which it established the keys.) We feel

that in many academic works the relevance of such references could be more clearly appreciated. The formal version of our observation is that session or key references are a prerequisite of sound composition results (as in [BFWW11]). Sound composition with other protocols plays a pivotal role also in the Universal Composability (UC) framework [Can01], and we are not surprised to see that the concept of a session reference emerges most clearly in the UC-related model of [KS05].

Also related to composition is the requirement of explicitly (and publicly) communicating session and key references, member lists, and information like the instance status: If a security model does not make this information readily available to an adversary, a reductionist security argument cannot use such information without becoming formally, and in many cases also effectively, invalid.

Finally, we emphasize that some GKE protocols allow for the concurrent execution of incompatible group operations (e.g. the concurrent addition and removal of a participant) so that different participants might derive keys with different understandings of whom they share it with. This indicates that the Designated Members category in Table 6.1 is quite important as well.

6.2.5 Our Syntax Proposal

We now turn to our syntax proposal that achieves all desirable properties from the above comparison (see Table 6.1). It is important to note that, in contrast to our *party-centric* perspective in the comparative systematization of this chapter, we design our model with an *instance-centric* view. That means, we here consider *instances* as the active entities in group key exchange and *parties* as only the passive static key-storage in authenticated GKE to which distinct groups of instances have joint access, respectively. We discuss the perspectives on the relation between instances and parties in more details in Section 6.5.1.

A GKE protocol G is a quadruple of algorithms $\mathsf{G} = (\mathrm{gen}, \mathrm{init}, \mathrm{exec}, \mathrm{proc})$ that generate authentication values, initialize an instance, execute

operations according to protocol-dependent commands, and process incoming ciphertexts received from other instances. In order to highlight simplifications that are possible for unauthenticated GKE, we indicate parts of the definition with gray marked boxes that are only applicable to the authenticated case of GKE.

We define GKE protocol G over sets $\mathcal{PAU}$, $\mathcal{SAU}$, $\mathcal{IID}$, $\mathcal{S}$, $\mathcal{CMD}$, $\mathcal{C}$, $\mathcal{K}$, and $\mathcal{KID}$ where $\mathcal{PAU}$ and $\mathcal{SAU}$ are the public and secret authenticator spaces, respectively (e.g., signing and verification key spaces, or public group identifier and symmetric pre-shared group secret spaces, etc.), $\mathcal{IID}$ is the space of identifiers that serve as instances' references, $\mathcal{S}$ is the space of instances' local secret states, $\mathcal{CMD}$ is the space of protocol-specific commands (that may include other instances' reference strings from $\mathcal{IID}$) to initiate operations in a session (such as adding users, etc), $\mathcal{C}$ is the space of protocol ciphertexts sent among instances, $\mathcal{K}$ is the space of group keys, and $\mathcal{KID}$ is the space of key identifiers that refer to computed group keys.

The key generation algorithm gen generates a pair of public and secret authenticator $(pau, sau) \in \mathcal{PAU} \times \mathcal{SAU}$; note that this pair is not necessarily linked cryptographically. Algorithm init takes as input a user-chosen instance identifier $iid \in \mathcal{IID}$ and outputs a fresh (secret) instance state $st \in \mathcal{S}$; note that, although exec and proc could implicitly initialize the state internally, we explicitly treat the state initialization for clarity reasons. With algorithm exec an instance can initiate the execution of an operation in a group (e.g., adding/joining/leaving/removing instances) by taking as input the current instance state $st \in \mathcal{S}$, an according command $cmd \in \mathcal{CMD}$ (potentially including affected instances' identifiers), and, optionally, a secret authenticator $sau \in \mathcal{SAU}$. The direct output is only the invoking instance's new state $st \in \mathcal{S}$; we handle outputs to the network and to the user explicitly below. To process an incoming ciphertext $c \in \mathcal{C}$, algorithm proc takes, in addition to it, instance state $st \in \mathcal{S}$ and, optionally, secret authenticator $sau \in \mathcal{SAU}$, and either outputs updated state $st \in \mathcal{S}$ or a rejection symbol. Accordingly, shortcut notations for these algorithms are

		gen	$\rightarrow_{\$}$	$\mathcal{PAU} \times \mathcal{SAU}$
$\mathcal{IID}$	$\rightarrow$	init	$\rightarrow_{\$}$	$\mathcal{S}$
$\mathcal{SAU} \times \mathcal{S} \times \mathcal{CMD}$	$\rightarrow$	exec	$\rightarrow_{\$}$	$\mathcal{S}$
$\mathcal{SAU} \times \mathcal{S} \times \mathcal{C}$	$\rightarrow$	proc	$\rightarrow_{\$}$	$\mathcal{S} \cup \{\perp\}$

Interfaces for algorithms In contrast to previous literature we treat communication to upper layer applications and to the underlying network infrastructure via interfaces that are provided by the environment in which a protocol runs rather than via direct return values. Thereby interfaces towards external protocols are explicitly separated from return values that serve as interfaces between algorithms of the primitive itself. Thus, each of the above algorithms can call the interfaces below (to send ciphertexts or report keys):

- $\mathcal{IID} \times \mathcal{C} \rightarrow \mathsf{snd}$ takes ciphertexts (and the calling instance's identifier) and sends these ciphertexts to the network such that they are potentially delivered to and processed by other instances.
- $\mathcal{IID} \times \mathcal{KID} \times \mathcal{K} \rightarrow \mathsf{key}$ takes keys with their key identifier (and the calling instance's identifier) and provides these keys to upper layer protocols.

Functions on objects We assume that the following functions can be computed on key identifiers, ciphertexts, or references to instances, respectively (for obtaining context information thereof; the advantage of this approach is that one can individually add and remove context information for more specific models at will). We note that the set of members is neither session-specific nor instance-specific but information that is attached to the context of each computed key. Since there might be keys that are computed for different sets of members in parallel (e.g., due to concurrently initiated conflicting membership operations), a single variable in an instance cannot express this information. One can interpret the notation below as getter-functions from object-oriented programming.

- $\mathcal{KID} \rightarrow \mathsf{mem} \rightarrow \mathcal{P}(\mathcal{IID})$ derives the identifiers of instances that are designated to be able to compute the referred key with their respective protocol execution.

- $\mathcal{C} \rightarrow \mathsf{r} \rightarrow \mathcal{P}(\mathcal{IID})$ derives identifiers of instances who are designated to receive the respective ciphertext.
- $\mathcal{IID} \rightarrow \mathsf{pau} \rightarrow \mathcal{PAU}$ derives the public authenticator of an instance from its identifier in the authenticated setting.

For further clarifications, we again refer to our discussion on instance- versus party-centric perspectives in Section 6.5.1.

6.3 Communication Models

The high flexibility in communication (i.e., interaction among participants) in a GKE protocol execution creates various problems for modeling and defining security of GKE: Firstly, tracing participants of a single global session is a non-trivial (if not highly complex) but very important issue. Nearly all considered GKE models trace communication partners differently and there exists an even wider variety of *partnering predicates* (aka. matching mechanisms) in the two-party key exchange literature that (aim to) fulfill this task. Secondly, normatively defining valid executions of a GKE protocol (versus invalid ones) in order to derive correctness requirements for them is not trivial for a generic consideration of GKE protocols. We note that only five out of the twelve considered models even define correctness. In the following we discuss partnering and correctness notions of the analyzed models.

6.3.1 Partnering

Partnering has served many different, somewhat independent purposes in (group) key exchange security models. In security experiments where an adversary trying to break a challenge key can also reveal 'independently' established keys, partnering is used to (1) determine, for a group key k of a challenge instance id, which other instances may also have computed the same key k to forbid the adversary trivially learning the challenge key through revealing id's partner instances' keys k (i.e., forbid trivial attacks). As such, the partnering predicate must include *at least* those instances that necessarily computed the

same key (e.g., group members), but it can be extended to further instances (such that the adversary is artificially weakened), for example, to allow for more efficient GKE constructions.

Partnering is sometimes used in security definitions of *explicitly authenticated* GKE to (2) identify successful active attacks against the authentication of an established key k if k was computed by an instance id without there existing partner instances at every designated group member. Consequently, the partnering predicate must include *at least* those instances belonging to designated members of a computed key, otherwise it is trivial to break authentication. But the predicate should not be extended to further instances, as actual attacks against authentication might go undetected, if partnering is used for this purpose.

Finally, partnering is (sometimes) used to (3) identify instances expected to compute the same key (i.e., define correctness requirements). In this case, the partnering predicate must include *at most* those instances that are required to compute the same key.

As these three purposes are at most loosely dependent on each other (if at all), defining them via one unified notion can lead to problems.[9]

Partnering also plays a crucial role in (4) the generic composability of (group) key exchange with other primitives: Brzuska et al. [BFWW11] show that a publicly computable partnering predicate is sufficient and (in some cases) even necessary for proving secure the composition of a symmetric key application with keys from an AKE protocol. (Although they consider two-party key exchange, the intuition is applicable to group key exchange as well.)

[9] During the research for this chapter, we found two recent papers' security definitions for two-party authenticated key exchange that, due to reusing the partnering definition for multiple purposes, cannot be fulfilled: Li and Schäge [LS17] and Cohn-Gordon et al. [CCG+19] both require in their papers' proceedings version for authentication that an instance only computes a key if there exists a partner instance that also computed the key (which is impossible as not all/both participants compute the key simultaneously). Still, the underlying partnering concept suffices for detecting reveals and challenges of the same key (between partnered instances). We informed the authors about this issue during the writing of this chapter.

Partnering/Matching/...	GKE specific	[BCPQ01]	[BCP02b]	[KY03]	[KS05]	[GBG09]	[CCG+18]	Chapter 5	[ACDT19]	[ACC+19b]	[BCP01]	[BCP02a]	[KLL04]	[YKLH18]	Desirable	Our model
Defined?	○	●	●	●	●	●	●	○	⊙	○	●	○	◕	●	●	●
Generic ● or protocol-specific ○	○	●	○	●	●	●	●	-	●	-	●	-	●	●	●	●
Normative/**P**recise/Retrospect. **V**ariable	○	**N**	-	**N**	**N**	**N**	**N**	-	**N**	-	**N**	-	**V**	**V**	**P**	**P**
⌞ Tight ● (vs. loose ○)	○	●	-	●	○	○	○	-	●	-	●	-	-	-	-	-
Publicly derivable	○	○	●	⊙	○	⊙	○	-	●	-	○	-	○	○	●	●
Components included in partnering predicate:																
Transcript																
Matching transcripts	○	⊙	⊙	●	○	○	○	-	●	-	⊙	-	○	○	○	○
Sequence of matching transcripts	●	●	⊙	○	○	○	○	-	○	-	●	-	○	○	○	○
Identifiers																
Group identifier	●	○	○	⊙	⊙	●	○	-	○	-	○	-	●*	●		○
Key identifier	○	○	○	○	○	○	○	-	○	-	○	-	●*	○		●
Externally input identifier	○	○	○	○	●	○	○	-	○	-	○	-	○	○		○
Group Key																
Whether partners computed a key	○	●	○	◔	●	●	●	-	○	-	●	-	○	○	⊙	⊙
Whether group computed a key	●	◔	○	○	○	○	○	-	○	-	◔	-	○	○		○
Whether partners computed same key	○	○	○	○	○	○	●	-	○	-	○	-	○	○	●	⊙
Members of the group	●	○	○	●	●	●	○	-	○	-	○	-	●	●	●	⊙

Table 6.2: Partnering Definitions. Notation: ●: yes, ⊙: implicitly, ◕: almost, ◔: partially, ○: no, -: not applicable.

Our Consideration of Partnering Predicates We consider the tracing of jointly computed equal keys ((1) above) as the core purpose of the partnering predicate, focusing on defining which keys can be revealed by an adversary. If the predicate is defined precisely (i.e., it exactly catches the set of same keys that result from a common global session) and is publicly derivable, it also allows for generic compositions of group key exchange with other primitives ((4) above), which we also consider indispensable.

It is thereby important to overcome a historic misconception of partnering: for either of the two above mentioned purposes not the *instances* that compute keys are central for the partnering predicate but the *keys themselves* and the *contexts* for which they are computed. In two-party key exchange, the context of a key is defined by its global session which itself is defined by its two participating instances. In multi-stage key exchange, keys are computed in consecutive stages of

a protocol execution. Hence, the context can be determined by the two participating instances in combination with the current (consecutive) stage number. However, in group key exchange—especially if we consider dynamic membership changes—the context of a key is not defined consecutively anymore: due to parallel, potentially conflicting changes of the member set in a protocol execution, it is not necessary that all instances, computing multiple keys, perform these computations in the same order. Consequently, partnering is not a linear, *monotone predicate* defined for instances but an *individual predicate* for each computed group key that reflects its individual context. This context can be protocol-dependent and may include the set of designated member instances, a record of operation by which its computation was initiated, etc. We treat the context information of group keys as an explicit output of the protocol execution also for supporting the use of these keys in upper layer applications (cf. Table 6.1).

Models Without Partnering Definitions Four models do not **define** a partnering predicate at all. In one of these, [ACDT19], a partnering predicate is implicit within their correctness definition. Three of these have no need of partnering since they restrict to (quasi-) passive adversaries ([ACDT19] and our model from Chapter 5) or do not allow adversarial reveal of group keys explicitly [ACC+19b], however by not defining a partnering predicate, they do not allow for generic composition with other primitives. [BCP02a] seemingly rely on an undefined partnering predicate, using the term 'partner' in their freshness definition but not defining it in the paper. [KLL04] defines a partnering predicate of which two crucial components (group and key identifier; see the asterisk marked items in Table 6.2) are neither defined generically nor defined for the specific protocol that is analyzed in it.

Generality of Predicates A partnering predicate can be **generic** or **protocol-specific**. From the considered models, only one has a predicate explicitly tailored to the construction. But many of the

generic partnering predicates involve values that are not necessarily part of all GKE schemes (e.g., group identifiers, externally input identifiers, etc.); a sufficiently generic partnering primitive should be able to cover a large class of constructions.

Character of Predicates Generic partnering predicates can be **normative**, **precise**, or **retrospectively variable**.

Normative predicates define objective, static conditions under which contexts of keys are declared partnered independent of whether a particular protocol, analyzed with it, computes equal keys under these conditions. This has normative character because protocols analyzed under these predicates must implement measures to let contexts that are—according to the predicate—declared unpartnered result in the computation of independent (or no) keys. As almost all security experiments allow adversaries to reveal keys that are not partnered with a challenge key (see Section 6.4), protocols that do not adhere to a specified normative predicate are automatically declared insecure. These predicates can hence be considered as (hidden) parts of the security definition.

The class of normative predicates can further be divided into **tight** and **loose** ones. **Tight** predicates define only those contexts partnered that result from a joint protocol execution not being attacked by active adversaries. This corresponds to the idea of *matching conversations* being the first tight predicate from the seminal work on key agreement by Bellare and Rogaway [BR94]. Two instances have matching conversations if each of them received a non-empty prefix of, or exactly the same as, what their peer instances sent—resulting in partnered contexts at the end of their session. Matching conversations are problematic for the GKE setting for two reasons. First, achieving security under matching conversations necessitates *strongly unforgeable* signatures or messages authentication codes (when being used to authenticate the communication transcript). Second, lifting matching conversations directly and incautiously to the group setting, as in [KY03], requires all communication in a global session

to be broadcast among all group members so each can compute the same transcript—inducing impractical inefficiency for real-world deployment. If the model's syntax generically allows to (partially) reveal ciphertexts' receivers, as in [ACDT19], pairwise transcript comparison does not require all ciphertexts to be broadcast but the strong unforgeability for authenticating signatures or MACs remains unnecessarily required. Several models [BCPQ01, BCP01] circumvent the necessity of broadcasting all group communication in a matching conversation-like predicate, although their syntax does not reveal receivers of ciphertext: they define two instances and their contexts as partnered if there exists a sequence of instances between them such that any consecutive instances in this sequence have partnered contexts according to matching conversations. (This still needs strongly unforgeable signatures and MACs, however.)

A **loose** partnering predicate is still static but declares more contexts partnered than those that inevitably result in the same key due to a joint, unimpeded protocol execution. This may include contexts of instances that actually did not participate in the same global session, or that did not compute the same (or any) key. An example for loose partnering predicates is *key partnering* [CCG+18] which declares the context of a key as the value of the key itself, regardless of whether it is computed due to participation in the same global session. Clearly, two instances that participated in two independent global sessions (e.g., one global session terminated before the other one begun) should intuitively not compute keys with partnered contexts even if these keys equal. Forbidding the reveal of group keys of intuitively unpartnered contexts results in security definitions that declare protocols 'secure' that may be intuitively insecure. On the other hand, partnering predicates that involve the comparison of a protocol-dependent [GBG09] or externally input group identifier are loose because equality of this identifier means being partnered but does not imply the computation of an equal (or any) key.

A **precise** partnering predicate exactly declares those contexts as partnered that refer to equal keys computed *due to* the participation in the same global session. Hence, the conditions for being partnered

are not static but depend on the respectively analyzed protocol. As a response to the disadvantages of normative partnering (and in particular tight matching conversations), Li and Schäge [LS17] proposed *original-key partnering* as a precise predicate for two-party key agreement: two instances have partnered contexts if they computed the same key, due to participating in a global session, that they would also have computed (when using the same random coins) in the absence of an adversary. As of yet, there exists no use of original-key partnering for the group setting in the literature, and we discuss drawbacks of this form of precise predicate with respect to the purpose of partnering below.

Variable predicates are parameterized by a customizable input that can be specified individually for each use of the model in which they are defined. Hence, these predicates are neither statically fixed nor determined for each protocol (individually) by their model, but can be specified ad hoc instead. As a result, a cryptographer, using a model with a variable predicate (e.g., when proving a construction secure in it), can define the exact partnering conditions for this predicate at will. The main drawback is that different instantiations of the same variable predicate in the same security model can produce different security statements for the same construction. We consider this ambiguity undesirable. Both group identifier and key identifier are left undefined in [KLL04] so they are effectively variable; in [YKLH18] the group ID is outsourced and thus left effectively variable.

Public Derivability of Predicates A partnering predicate can and—in order to allow for generic compositions—should be **publicly derivable**. That is, the set of partnered contexts should be deducible from the adversarial interaction with the security experiment (or, according to Brzuska et al. [BFWW11], from the communication transcript of all instances in the environment). Only four models considered achieve this as listed in Table 6.2; ⊙ here refers to the implicit ability to observe whether group keys are computed. Partnering in all remaining models involves values in instances' secret states. Original-

key partnering [LS17] (for two-party key exchange) is the only known precise predicate but it is not publicly computable as it depends on secret random coins.

Components of Predicates The lower part of Table 6.2 lists the various parameters on which partnering predicates we consider are defined. These parameters include: the **transcript** of communications, protocol-specific **identifiers**, **external inputs**, the computed **group key**, the set of **members of the group**, etc. There are many other potential parameters as well (cf., two-party key exchange literature).

The two main purposes of partnering ((1) forbidding trivial attacks and (4) allowing for generic composition) use the partnering predicate to determine which keys computed during a protocol execution are meant to be the same and in fact equal (i.e., whether they share the same context). Consequently, an ideal partnering predicate should depend on the context that describes the circumstances under which (and if) the group key is computed. As only for some protocols (e.g., optimal secure ones; cf. our security notions in chapters 3 and 4) it is reasonable that the entire communicated transcript primarily determines the circumstances (i.e., the context) under which a key is computed, we consider it unsuitable for defining partnering generically.

6.3.2 Correctness

Correctness is a quality of protocols that describes the functional guarantees one can expect from their execution. Before discussing these guarantees and the requirements under which one can expect them, we first discuss the role of correctness definitions in models that are aimed for the analysis of another (potentially independent) quality: security.

Safety and Liveness While usually being defined unified in cryptographic literature, *correctness* is often alternatively considered as

the combination of distinct *safety* and *liveness* predicates in other research fields (such as distributed computing). Thereby safety captures consistency guarantees and liveness declares conditions under which actual functionality is guaranteed. Intuitively, liveness expresses under which conditions the protocol execution computes an output (e.g., if the adversary remains passive and the protocol terminates, then a key is computed by all execution participants), and safety expresses which conditions this output must fulfill when it is computed, without requiring that it is ever computed (e.g., if a key is computed by some participants during a session, then it must be the same key for all of them). After explaining the problems with liveness definitions in interactive group protocols, we discuss which of these two components (safety and liveness) are important for defining security.

Liveness in Interactive Group Protocols For most cryptographic primitives, correctness in the form of liveness is defined by requiring a specific output behavior for a specific execution schedule. In the simplest (non-interactive) case, the nested execution of several algorithms is required to produce a certain output (e.g., for encryption schemes, the decryption of the encryption of a message must produce the message again; for signature schemes, the verification of a message with a signature that was computed for this message must accept; etc). Similarly, for two-party key exchange (if correctness is defined) the honest execution of the protocol is usually required to establish the same key for both execution participants (see e.g., [BR94]).

For protocol executions (i.e., sessions) with multiple participants that can (all) actively influence the output of this execution (e.g., dynamic GKE in which all group members can initiate membership changes), there may exist multiple different execution schedules that results in the same output (e.g., the same group can compute the same key independent of the order of membership changes). Simultaneously, there may not exist only one 'correct' output for each specific execution schedule. (Consider, for example, a dynamic GKE session in which two group members concurrently initiate conflicting changes

of membership. The resulting set of members and the group key, output by participating members, may differ for different GKE schemes, or even for different executions of the same GKE scheme.) Finally, as argued in Section 6.2, we consider it undesirable to explicitly model specific operations (and their impact on the session) in GKE syntax definitions. Specifying the exact output of specific execution schedules as part of a correctness definition is thereby undesirable as well. It is, consequently, complicated (if not impossible) to formulate a complete and static definition of execution schedules with their required output in a liveness definition for generic GKE protocols.

Correctness	GKE specific	[BCPQ01]	[BCP02b]	[KY03]	[KS05]	[GBG09]	[CCG+18]	Chapter 5	[ACDT19]	[ACC+19b]	[BCP01]	[BCP02a]	[KLL04]	[YKLH18]	Desirable	Our model
Defined	○	○	○	●	●	●	○	●	●	○	○	○	○	●	●	●
Requirements																
Honest transcript delivery	○	-	-	⊙	●	●	-	●	●	-	-	-	-	○		○
Two instances are partnered	○	-	-	●	○	○	-	○	⊙	-	-	-	-	●	⊙	⊙
All group instances are partnered	●	-	-	○	○	○	-	○	○	-	-	-	-	○		○
A key is computed	○	-	-	●	○	○	-	○	○	-	-	-	-	●	⊙	⊙
All participating instances share *gid*	○	-	-	○	●	○	-	○	○	-	-	-	-	○		○
Keys are partnered	○	-	-	⊙	○	○	-	○	⊙	-	-	-	-	⊙	●	●
Guarantees																
Same key	○	-	-	●	●	●	-	●	●	-	-	-	-	●	●	●
Non-zero key	○	-	-	●	○	○	-	○	○	-	-	-	-	○		○
Keys are partnered	○	-	-	⊙	⊙	●	-	○	⊙	-	-	-	-	○		⊙

Table 6.3: Correctness Definitions. Notation: ●: yes, ⊙: implicitly, ◕: almost, ◔: partially, ○: no, -: not applicable.

Correctness and Security Intuitively, correctness and security could be considered independent qualities: a scheme can be correct but insecure, incorrect but secure, correct and secure, or neither correct nor secure. Indeed, 'lazy' (i.e., incorrect) schemes not doing anything trivially (can be adapted to) reach most security properties immediately. Thereby the question may arise why one needs to consider correctness in a security analysis at all.

We observe that there exist security definitions that cannot be

achieved by schemes that provide certain correctness guarantees and consequently there exist correctness definitions that schemes cannot achieve when also providing certain security guarantees. Hence, for demonstrating the usefulness of a security definition, either a correctness definition (including liveness guarantees) that is (believed to be) compatible with this security should be provided. Alternatively, a construction accordingly secure, implicitly showing its correctness guarantees, should be provided along with this security definition. Beyond demonstrating usefulness, many natural security definitions implicitly or explicitly rely on an underlying safety definition. In Section 6.3.3 we discuss this idea of *security up to correctness* [RZ18] in more details.

We conclude that defining correctness in definitions of security *can be desirable* in the form of *liveness* for demonstrating that the security can be met by useful constructions, and *is desirable* in the form of *safety* for guiding restrictions of the adversary in the security experiment. As a definition of security normally comes with an analysis of a (useful) protocol, the definition of liveness can usually be neglected. Furthermore, due to the sketched problems with (complete and static) liveness definitions, we focus on safety guarantees in the following.

A Clean Definition of Safety The only functionality, GKE protocols should provide generically, is the establishment of group keys. Thereby keys should be the same if they were output by participants of a joint session and meant to be established as a shared group key. The generic mechanism for tracing commonly computed group keys is the partnering predicate. Consequently, partnered keys (or keys of partnered instances) should be equal in order to fulfill safety (see Table 6.3).

As our syntax from Section 6.2.5 allows upper layer protocols to identify keys (with the key identifier), it is actually not necessary that two session participants compute two distinct group keys in the same order. Partnering of instances, however, seems to be a linearly sequential property (if two instances were partnered once and their

partnerships is disrupted, they will not become partners again). Partnering of group keys is, in contrast, a time-independent property (if two keys are partnered, they will remain partnered forever). As a consequence, we consider keys (as opposed to instances) being partnered desirable as the requirement of the safety definition.

Discussion of Models As it can be seen in Table 6.3, two definitions [KY03, YKLH18] match our idea of requirements for correctness (note that both define partnering w.r.t. instances and [KY03] require honest delivery for being partnered). In addition to requiring key equality as safety guarantee, [KY03] furthermore demand the computation of a non-trivial *real* key.

The remaining three correctness definitions require an honest protocol execution (satisfying their explicit or implicit partnering predicates) for the computation of same keys. The above described problem of declaring an execution *honest* (and requiring a certain output from it) is resolved in these three definitions as follows: The former two models [KS05, GBG09] solely capture static GKE (in which the execution schedule can be predetermined) and the latter one forbids the concurrent processing of conflicting operations in groups such that a fixed, shared output can be expected.

6.3.3 Safety and Security

Based on the idea that an adversary in a security experiment only needs to be restricted when attacking a correct scheme, the idea of *security up to correctness* (more precisely "indistinguishability up to correctness") was formally proposed by Rogaway and Zhang [RZ18]. Their approach for defining security for a given correctness definition is intuitively as follows: in the security experiment adversaries are given full power over the execution of a primitive simulated by a challenger, potentially including access to special oracles that provide secrets of instances involved in the execution. The challenger then restricts this power by "silencing" (i.e., suppressing) only adversarial queries for which the challenger's response is already determined by

the correctness definition. This restriction prevents the trivial solution of embedded challenges in the security experiment (e.g., if correctness requires partnered keys k', k^* to equal $k' = k^*$, then if k^* is a challenge, revealing k' determines the challenge's solution). The major advantage of this approach is reduced ambiguity in defining security. Note that one can still define oracles for adversarial access to secrets freely.

It is immediate that the approach of Rogaway and Zhang [RZ18] requires the definition of correctness for obtaining a definition of security. However, the challenger, when silencing oracle queries based on their determination, does not need to know in advance *when* the protocol computes an output (as the outputs of the protocol execution are directly observable for the challenger). The challenger only needs to know how a computed output relates to other values. This relation is expressed by the definition of safety already.

We note that the overall idea behind *security up to correctness*—silencing oracles or penalizing queries to them, based on the safety guarantees one expects—is not only the basis of this formal definition process but also the intuition behind most informal, manual approaches for defining security.

6.3.4 Our Partnering and Correctness Proposals

We here shortly describe the details of our partnering and correctness definitions, the latter being based on the game from Figure 6.1.

Definition 3 (Partnering) *Two keys k_1, k_2 output as tuples (kid_1, k_1) and (kid_2, k_2) via interface* key *are partnered iff $kid_1 = kid_2$.*

Our partnering predicate, hence, defines keys with the same explicitly output context *kid* partnered.

Definition 4 (Safety) *A GKE protocol* G *is safe if for all adversaries $\mathcal{A}$ against game* FUNC *according to Figure 6.1 it holds that* $\Pr[\text{FUNC}_{\mathsf{G}}(\mathcal{A}) = 1] = 0$.

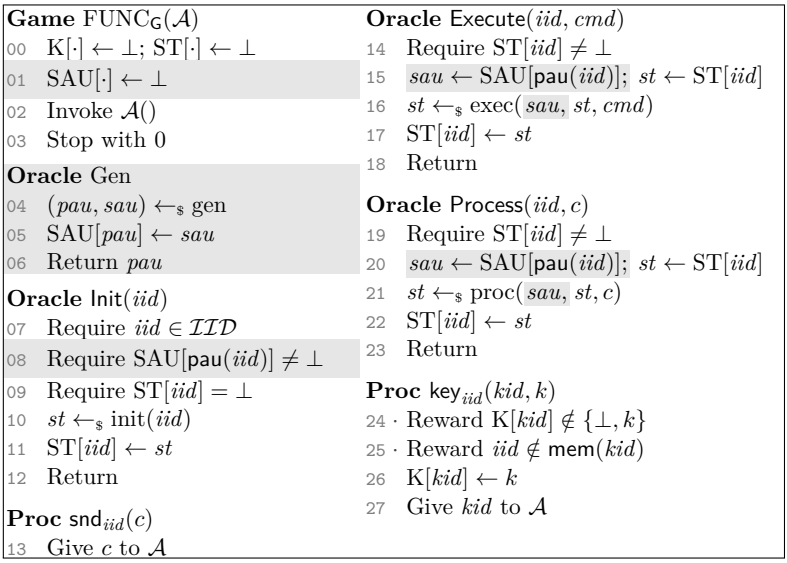

Figure 6.1: FUNC game of GKE describing correctness in the form of safety. Gray marked code is only applicable in the authenticated setting. Lines marked with '·' at the left margin highlight safety requirements. For an explanation of the used variables see Table 6.4.

We declare a GKE protocol incorrect if two computed non-trivial keys (i.e., $k \neq \perp$) with equal key identifiers differ (see Figure 6.1 line 24).

Beyond the core functionality of establishing group keys, our syntax allows upper layer protocols to derive the set of designated group members for each established group key. For this additional functionality we require that group keys, output by an instance via interface key, are actually designated to this instance (Figure 6.1 line 25). We emphasize that this requirement is indeed a property of safety (and not security) as it does not hinder any independent exposable instance to be able to compute the key internally without outputting it.

All remaining pseudo-code in Figure 6.1 only describes the necessary framework for executing the algorithms inside the oracles that are provided to adversaries. Consequently, these parts of the Figure are

irrelevant for understanding the definition and equally appear in our security experiment in Figure 6.2.

K	Array of computed group keys
ST	Array of instance states
SAU	Array of secret authenticators
CR	Set of corrupted or external authenticators
WK	Set of weak group keys
CH	Set of keys challenged for $\mathcal{A}$
CP	Set of keys already computed by an instance
TR	Transcript as queue of ciphertexts sent among instances

Table 6.4: Variables in figures 6.1 and 6.2.

6.4 Security Definitions

Although the actual definition of security is the core of a security model, there is no unified notion of 'security' nor agreement on how strong or weak 'security' should be—in part because different scenarios demand different strengths. Thus we do not aim to compare the strength of models' security definitions, but do review clearly their comparable properties. We focus on the desired security goals, adversarial power in controlling the victims' protocol execution, and adversarial access to victims' secret information. We do not compare the conditions under which adversaries win the respective security experiments (aka. 'freshness predicates', 'adversarial restrictions', etc.) as this relates to the models' 'strength', but we do report on characteristics such as forward-secrecy or post-compromise security.

Security Goals The analyzed models primarily consider two independent security goals: secrecy of keys and authentication of participants.

Secrecy of keys is in all models realized as **indistinguishability** of actually established **keys** from random values, within the context

Security	GKE specific	[BCPQ01]	[BCPQ02b]	[KY03]	[KS05]	[GBG09]	[CCG+18]	Chapter 5	[ACDT19]	[ACC+19b]	[BCP01]	[BCP02a]	[KLL04]	[YKLH18]	Desirable	Our model
Security Goals																
Key indistinguishability	○	●	●	●	●	●	●	●	●	●	●	●	●	●	●	●
⌞ Multiple challenges	○	○	○	○	○	○	○	●	●	○	○	○	○	○	●	●
Explicit authentication	○	●	○	○	●	●	○	○	○	○	●	○	◔	○		○
Adversarial Protocol Execution																
All algorithms	○	●	●	●	●	●	●	○	◕	◕	●	●	●	●	●	●
Instance specific	●	●	●	●	●	●	●	○	●	●	◔	◔	◔	●	●	●
Concurrent invocations	●	●	●	●	●	●	●	◕	◔	●	○	○	○	●	●	●
Active communication manipulation	○	●	●	●	●	●	◕	○	○	◔	●	●	●	●	●	●
Adversarial Access to Secrets																
Corruption of involved parties' secrets	○	●	○	●	●	●	●	-	-	○	●	●	●	●	●	●
⌞ After key exchange	○	●	-	●	●	●	●	-	-	-	●	●	◕	●	●	●
⌞ Before key exchange	○	○	-	○	○	●	●	-	-	-	○	○	○	●		●
Corruption of independent parties' secrets	○	●	○	●	●	●	●	-	-	○	●	●	●	●	●	●
⌞ Always	○	●	-	●	●	●	●	-	-	-	○	○	●	●	●	●
Exposure of involved instances' states	○	○	○	○	●	●	◕	●	●	●	○	●	○	●	●	●
⌞ After key exchange	○	-	-	-	●	●	◕	●	●	●	-	●	-	●	●	●
⌞ Before key exchange	○	-	-	-	○	○	●	●	●	●	-	○	-	○		○
Exposure of independent instances' states	○	○	○	○	●	●	●	-	●	●	○	●	○	●	●	●
⌞ Always	○	-	-	-	●	●	●	-	●	●	-	○	-	○	●	●
Reveal of independent group keys	○	●	●	●	●	●	●	●	●	⦿	●	◔	◔	●	●	●
⌞ Always	○	●	●	●	●	●	●	●	●	⦿	●	◔	◔	○	●	●

Table 6.5: Security Definitions. Notation: ●: yes, ⦿: implicitly, ◕: almost, ◔: partially, ○: no, -: not applicable.

of an experiment in which the adversary controls protocol executions. During the experiment, the adversary can query a challenge oracle that outputs either the real key for a particular context or a random key; a protocol is *secure* if the adversary cannot distinguish between these two. Only one model allows adversaries to query the **challenge** oracle **multiple** times; all others allow only one query to the challenge oracle, resulting in an unnecessary and undesirable tightness loss in reduction-based proofs of composition results.

Key indistinguishability against active adversaries already implies *implicit authentication* of participants. That means keys computed in a session must diverge in case of active attacks that modify communications. Some models require **explicit authentication**: that the protocol explicitly rejects when there was an active attack. However,

the value of explicit authentication in GKE, or even authenticated key exchange broadly, has long been unclear [Sho99]: GKE is never a standalone application but only a building block for some other purpose, providing keys that are implicitly authenticated and thus known only to the intended participants. If the subsequent application aims for explicit authentication of its payload, the diverging of keys due to implicit authentication can be used accordingly.

Adversarial Protocol Execution To model the most general attacks by an adversary, the security experiment should allow adversaries to setup the experiment and control **all** victims' invocations of protocol **algorithms** and operations; all models considered (except for our restricted one from Chapter 5) do so. However, in two models the adversary can setup only one group during the entire security experiment (◕); this again introduces a tightness loss in the number of groups for composition results, and means that the use of long-term keys by parties across different sessions, as defined by [ACC+19b], cannot be proven secure in the respective model.

Most models allow for **instance-specific** scheduling of invocations. But four models (◔/○) require the adversary schedules algorithm and operation invocations for all affected instances together; diverging and (for ◔) concurrent invocations cannot be scheduled in these four models. In practice this restriction means that some form of consensus is required (e.g., a central delivery server). While algorithms and operations can be **invoked concurrently** in [ACC+19b], this model allows only one of the resulting concurrently sent ciphertexts to be delivered to and processed by the other participants of the same session; this similarly requires some consensus mechanism.

An **active** adversary who **modifies communication** between instances is permitted in almost all models. However, [CCG+18] forbid active attacks during the first communication round, [ACC+19b] only allow adversaries to inconsistently forward ciphertexts but neither reorder nor manipulate them, and [ACDT19] as well as our model from Chapter 5 require honest delivery of the transcript. For the deploy-

ment of protocols secure according to the latter three models, active adversaries must be considered impractical or authentication mechanisms must be added.

Adversarial Access to Secrets GKE models allow the adversary to learn certain secrets used by simulated participants during the security experiment. Below we discuss the different secrets that can be learned and the conditions under which this is allowed. We neglect adversarial access to algorithm invocations' random coins in our systematization as only three models consider this threat in their security experiments [CCG+18, ACDT19, ACC+19b].

Corruption of party secrets models a natural threat scenario where parties use static secrets to authenticate themselves over a long period. Corruption is also necessary to model adversarial participation in environments with closed public key infrastructure (cf. Section 6.2), allowing the adversary to impersonate some party. Table 6.5 shows which models allow for corruptions of party secrets *after* and *before* the exchange of a secure group key (i.e., *forward-secrecy* and *post-compromise security*, respectively), and corruptions of independent parties anytime. In [ACDT19] parties do not maintain static secrets so corruption is irrelevant. Two other models do have parties with static secrets but do not provide an oracle for the adversary to corrupt them.[10] Due to imprecise definitions, [KLL04] partially forbids corruptions of involved parties even after a secure key was established, and two other models even forbid corruptions of independent parties before an (independent) secure group key is established. Only three models treat authentication as the sole purpose of party secrets, defining precise conditions that allow corruptions before and after the establishment of a secure group key. As secrecy of a group key should never depend solely on secrecy of independent parties' long-term se-

[10] Moreover, in [BCP02b, ACC+19b], party secrets cannot be derived via state exposures. Although [ACC+19b] allow the exposure of instance states, their syntax, strictly speaking, does not have a method for *using* party secrets in the protocol execution, even though their construction makes use of them (violating the syntax definition).

crets and *forward-secrecy* is today considered a minimum standard, we deem security despite later corruption of long-term secrets desirable.

Exposure of instance states is especially important in GKE because single sessions may be quite long-lived—such as months- or years-long chats—so local states may become as persistent as party secrets. In most security experiments that provide adversarial access to instance states, their exposure is not permitted before the establishment of a secure group key. Some of these models further restrict the exposure of independent instances' states (e.g., because they were involved in earlier stages of the same session). The three articles and our own model that consider *ratcheting* of state secrets allow adversarial access to these states shortly before and after the establishment of a secure group key. [CCG+18] model state expose through the reveal of random coins, which means an exposure at a particular moment reveals only newly generated secrets in the current state, not old state secrets. We consider the ability to expose states independent of and after the establishment of a group key desirable, and leave state exposure before establishment—post-compromise security—as a bonus feature.

The **reveal** of established **group keys** in the security experiment is important to show that different group keys are indeed independent. One motivation for this is that use of keys in weak applications should not hurt secure applications that use different keys from the same GKE protocol. The reveal of keys is furthermore necessary to prove implicit authentication of group keys. Reveals should also be possible to permit composition of key exchange with a generic symmetric key protocol [BFWW11]. Almost all models allow the reveal of different (i.e., unpartnered) group keys unlimitedly. As [BCP02a] and [KLL04] do not define partnering adequately (see Section 6.3.1), it cannot be assessed which group keys are declared *unpartnered* in their models. The adversary in [ACC+19b] is not equipped with a dedicated reveal oracle but since the security in this model is strong enough, the exposure of instance states suffices to obtain all keys without affecting unpartnered keys. [YKLH18] forbid the reveal of earlier group keys in the same session. As unpartnered keys should always be

independent we consider it desirable to allow their unrestricted reveal.

6.4.1 Our Security Proposal

We define security (see Figure 6.2) by allowing adversaries to interact with the scheme's algorithms via oracles Init, Execute, Process in the unauthenticated setting and additionally Gen in the authenticated setting, plus oracles to obtain secrets Expose, Reveal in the unauthenticated setting and additionally Corrupt in the authenticated setting, plus an oracle Challenge to obtain challenges. In the following we describe how we restrict the adversary (e.g., to prevent the trivial solving of challenges). We note that almost all pseudo-code in Figure 6.2 also appears in Figure 6.1 (building the infrastructure of the simulation by the challenger). Only the remaining parts are relevant for understanding restrictions of the adversary and the definition of security.

Restriction of the Adversary The simplest trivial attack that our security experiment forbids is:

1. A key must not be both revealed and queried as a challenge (lines 36,41,05).

We treat local state exposures and their effects as follows:

2. After an exposure all keys that can be computed by the exposed instance are declared weak (i.e., known to the adversary), if they have not already been computed by this exposed instance (lines 44,33).

We thereby require forward-secrecy (as previously computed keys are required to stay secure after an exposure) but not post-compromise security (as all future keys of this instance are declared insecure after an exposure) since we do not aim for an optimal security definition.

Finally, the treatment of active attacks against the transcripts in the unauthenticated setting is as follows:

3a) If a ciphertext from an unknown source (or from a known source in the wrong order) is processed without being rejected (lines 29-30,25) then all keys that can be computed by the processing instance are declared weak, if they have not already been computed by this processing instance (lines 26,33).

In the authenticated setting, the set of keys that are declared weak is reduced based on the set of corrupted authenticators. Authenticators are considered *corrupted* if they have not be generated by the challenger (lines 01,09; because thereby they are potentially adversarially generated) or if they have been honestly generated first but then corrupted (lines 01,09,47). As the impersonation of all uncorrupted authenticators should be hard in the authenticated setting, active attacks against the transcripts are treated as follows:

3b) If a ciphertext from an unknown source (or from a known source in the wrong order) is processed without being rejected (lines 29-30,25) then all keys that can be computed by the processing instance are declared weak, if they have not already been computed by this processing instance (lines 26,33) and they are marked to be computable by an instance with a corrupted authenticator (lines 26,01,09,47).

Definition 5 (Adversarial Advantage) *The advantage of an adversary* $\mathcal{A}$ *in winning game* KIND *from Figure 6.2 is* $\mathrm{Adv}^{\mathrm{kind}}_{\mathsf{G}}(\mathcal{A}) := |\Pr[\mathrm{KIND}^{1}_{\mathsf{G}}(\mathcal{A}) = 1] - \Pr[\mathrm{KIND}^{0}_{\mathsf{G}}(\mathcal{A}) = 1]|$.

Game $\mathrm{KIND}^b_{\mathsf{G}}(\mathcal{A})$
00 $\mathrm{K}[\cdot] \leftarrow \bot$; $\mathrm{ST}[\cdot] \leftarrow \bot$
01 $\mathrm{SAU}[\cdot] \leftarrow \bot$; $\mathrm{CR} \leftarrow \mathcal{PAU}$
02 $\mathrm{WK} \leftarrow \emptyset$; $\mathrm{CH} \leftarrow \emptyset$
03 $\mathrm{CP}[\cdot] \leftarrow \emptyset$; $\mathrm{TR}[\cdot][\cdot] \leftarrow \bot$
04 $b' \leftarrow_{\$} \mathcal{A}()$
05 · Require $\mathrm{WK} \cap \mathrm{CH} = \emptyset$
06 Stop with b'

Oracle Gen
07 $(pau, sau) \leftarrow_{\$} \mathrm{gen}$
08 $\mathrm{SAU}[pau] \leftarrow sau$
09 · $\mathrm{CR} \leftarrow \mathrm{CR} \setminus \{pau\}$
10 Return pau

Oracle $\mathsf{Init}(iid)$
11 Require $iid \in \mathcal{IID}$
12 Require $\mathrm{SAU}[\mathsf{pau}(iid)] \neq \bot$
13 Require $\mathrm{ST}[iid] = \bot$
14 $st \leftarrow_{\$} \mathrm{init}(iid)$
15 $\mathrm{ST}[iid] \leftarrow st$
16 Return

Oracle $\mathsf{Execute}(iid, cmd)$
17 Require $\mathrm{ST}[iid] \neq \bot$
18 $sau \leftarrow \mathrm{SAU}[\mathsf{pau}(iid)]$; $st \leftarrow \mathrm{ST}[iid]$
19 $st \leftarrow_{\$} \mathrm{exec}(sau, st, cmd)$
20 $\mathrm{ST}[iid] \leftarrow st$
21 Return

Oracle $\mathsf{Process}(iid, c)$
22 Require $\mathrm{ST}[iid] \neq \bot$
23 $sau \leftarrow \mathrm{SAU}[\mathsf{pau}(iid)]$; $st \leftarrow \mathrm{ST}[iid]$
24 $st \leftarrow_{\$} \mathrm{proc}(sau, st, c)$
25 · If $\nexists iid_s : c = \mathrm{TR}[iid_s][iid].dequeue()$
$\wedge st \neq \bot$:
26 · WK $\xleftarrow{\cup}$ $\{kid \in \mathcal{KID} \setminus \mathrm{CP}[iid] :$
$\exists iid_{cr} : \{iid, iid_{cr}\} \subseteq \mathsf{mem}(kid)$
$\wedge\, \mathsf{pau}(iid_{cr}) \in \mathrm{CR}\}$
27 $\mathrm{ST}[iid] \leftarrow st$
28 Return

Proc $\mathsf{snd}_{iid}(c)$
29 · For all $iid_r \in \mathsf{r}(c)$:
30 · $\mathrm{TR}[iid][iid_r].enqueue(c)$
31 Give c to $\mathcal{A}$

Proc $\mathsf{key}_{iid}(kid, k)$
32 $\mathrm{K}[kid] \leftarrow k$
33 · $\mathrm{CP}[iid] \xleftarrow{\cup} \{kid\}$
34 Give kid to $\mathcal{A}$

Oracle Reveal(kid)
35 Require $\mathrm{K}[kid] \neq \bot$
36 · WK $\xleftarrow{\cup}$ kid
37 Return $\mathrm{K}[kid]$

Oracle Challenge(kid)
38 Require $\mathrm{K}[kid] \neq \bot \wedge kid \notin \mathrm{CH}$
39 $k_0 \leftarrow \mathrm{K}[kid]$
40 $k_1 \leftarrow_{\$} \mathcal{K}$
41 · CH $\xleftarrow{\cup}$ kid
42 Return k_b

Oracle Expose(iid)
43 Require $\mathrm{ST}[iid] \neq \bot$
44 · WK $\xleftarrow{\cup}$ $\{kid \in \mathcal{KID} \setminus \mathrm{CP}[iid] :$
$iid \in \mathsf{mem}(kid)\}$
45 Return $\mathrm{ST}[iid]$

Oracle Corrupt(pau)
46 Require $\mathrm{SAU}[pau] \neq \bot$
47 · CR $\xleftarrow{\cup}$ $\{pau\}$
48 Return $\mathrm{SAU}[pau]$

Figure 6.2: KIND game of GKE modeling unauthenticated or authenticated group key exchange. '·' at the left margin of a line highlights mechanisms for restricting the adversary (e.g., to forbid trivial attacks). Almost all remaining parts of the game equally appear in game FUNC in Figure 6.1 and are less important for understanding the security definition. For an explanation of the used variables see Table 6.4.

Discussion of the Model With respect to the required security, our aim is only to give an example definition. As indicated before, we believe that optimal security for GKE is usually undesired and we are not under the illusion that there exists a unified definition of security on which the literature should or aims to agree on. Our contribution is instead that we provide a simple, compact, and precise framework that generically captures GKE and in which the restriction of the adversary (which essentially models the required security) can easily be adjusted. Thereby it achieves all properties that we declare desirable in our systematization of knowledge. To name only some advantages of our model: 1. it allows for participation of multiple instances per party per session, 2. it covers unauthenticated, symmetric-key authenticated, and public-key authenticated settings, 3. it imposes no form of key distribution mechanism on GKE constructions and their environment, 4. neither does it impose a consensus mechanism for unifying all session participants' views on the session (although they can be implemented on top), 5. it fits for any variant of protocol-specific membership operations, 6. thereby it formulates natural generic (programming) interfaces through the syntax, 7. it outputs the context of group keys along the group keys themselves to upper-layer applications, 8. thereby it allows for actual asynchronous protocol executions in which not all participants need to agree upon the same order of group key computations, 9. it defines partnering naturally via the context that the protocol itself declares for each group key, 10. it illustrates how a generic model can allow for protocol-dependent definitions of contexts for group keys, 11. thereby it respects the requirements of composition results [BFWW11], 12. it naturally gives adversaries in the security experiment full power in executing the protocol algorithms and determining their public inputs, 13. and it can easily express different strengths of security.

This model consequently fulfills its main purpose: demonstrating that the desired properties from our systematization framework do not conflict and can hence be achieved simultaneously.

6.5 Discussion

Our systematization of knowledge reveals some shortcomings in the GKE literature, stemming from a tendency to design a security model hand-in-hand with a protocol to be proven; such a model tends to be less generic, making specific assumptions about characteristics of the protocols it can be used for or the application environment with which it interacts. Sometimes the application environment appeared to be fully neglected. We have tried to revisit the underlying concepts of GKE and take into account the broad spectrum of requirements that may arise from the context in which a GKE protocol may be used, such as the type and distribution of authentication credentials of parties, how groups are formed and administered, and whether parties can have multiple devices in the same group. The goal is not to develop a single unified model of group key exchange security, but to support the development of models within the GKE literature that are well-informed by the principle requirements of GKE. Our prototype model demonstrates that these desirable properties of GKE can be satisfied within one generic model, with reduced complexity and increased precision.

Looking forward, group key exchange is on track for increasing complexity. There now exist prominent applications requiring group key exchange—group instant messaging, videoconferencing—and using a cryptographic protocol in a real-world setting invariably leads to greater complexity in modeling and design. Moreover, the desire for novel properties such as highly dynamic groups and post-compromise security using ratcheting, manifested in proposed standards such as MLS, make it all the more important to have a clear approach to modeling the security of group key exchange.

6.5.1 The Relation between Parties and Instances

Even with our systematic approach some semantic interpretations of modeling choices are not ultimately determined. One example that we discuss here is that it remains debatable whether *local instances*

or *parties* are the *actual participants* of sessions. In the **instance-centric perspective**, instances are *active* entities that may or may not represent *passive* parties. The term party would thereby only refer to static objects that embody authentication secrets whereas instances would be considered active entities that potentially make use of these static objects. In the **party-centric perspective**, instances are only realizations of parties' participation in sessions. Parties would thereby *actively* outsource the execution of algorithms, necessary for participating in sessions, to their instances. As this distinction may appear as quibble on a language level, we clarify technical differences below.

Instance-Centric Perspective We first consider a setting that, except for our own model from Chapter 5, neither of the publications, analyzed in this chapter, covers: unauthenticated GKE. In unauthenticated GKE the concept of parties remains unclear. While parties usually refer to permanent entities, participants of unauthenticated GKE session only temporarily exist during their participation. Additionally, parties are often linked to authentication secrets. According to this interpretation, parties do not exist in unauthenticated GKE such that only instances could be participants of sessions. Secondly, if instances are the actual participants of sessions, it can be reasonable to allow simultaneous participation of multiple instances per party in sessions. This can be a useful feature for multi-device settings (e.g., in instant messaging). Thirdly, GKE is never a primitive used by humans but a tool used by other cryptographic applications. GKE instances, executing the participation in a GKE session, may thereby be initiated by instances of the upper layer cryptographic application rather than by a central GKE party. Thereby the concept of a party collapses to some static information (e.g., authentication secrets) that these GKE instances use.

Party-Centric Perspective If parties are active participants of sessions (i.e., *members* thereof) trough their instances, participation

in a single session through multiple instances per party is contradictory. Therefore settings in which human users participate in a session with multiple devices would need to be modeled by declaring each device of a user an individual party. We note that sharing authentication secrets between these "device-parties" would be considered insecure in all analyzed models. Nevertheless, the concept of active parties and their realization through controlled instances intuitively describes the idea of (authenticated) entities using the GKE protocol for deriving group keys more comprehensibly than the instance-centric perspective. Even if GKE instances are locally initiated by distributed instances of the cryptographic application that uses the group keys, one can trace the initialization of these application instances back to a central active party. As application executions of the same party are in reality usually initiated and managed centrally, conceiving this central party as an active entity instead of its multiple local instances appears to be practical.

We believe that neither of both perspectives describes the essential truth and preferences for either of them may depend on the individual understanding of what *parties* and *instances* are in reality. Nevertheless, for modeling GKE one needs to commit to either of them, implicating the respective consequences described above. As all but one analyzed models adopt the party-centric perspective (see the first three rows in Table 6.1), the terminology in our comparison is chosen accordingly. However, since this intuitively restricts parties from participating in sessions with multiple of their instances, we chose the instance-centric perspective in our proposed model.

7
Conclusions and Outlook

The core of this thesis is the analysis of cryptographic definitions and constructions for components in modern messaging applications. The main technique of these components that we consider is the continuous updating of key material. Our consideration includes and affects many different more general aspects in cryptography that we contextualize after a short summary of the main chapters in this thesis.

7.1 Overview

In Chapter 3, we provide a systematic and, due to our holistic approach, potentially even fundamental study of continuous key exchange in the two-party setting. Instead of understanding the underlying idea of ratcheting from the construction-perspective, we leverage the approach by Bellare et al. [BSJ+17] that defines syntax, correctness, and security *naturally* with respect to what this cryptographic primitive *can* and potentially *should* provide in theory. Recall that, in this initial unidirectional variant, Alice and Bob jointly compute local states, and then Alice establishes keys for both users by sending ciphertexts to Bob. We redefine the corresponding security notion by Bellare at al. that impractically forbids adversarial exposures of Bob's local state. In our notion, both user states can permanently be exposed, and we require that these exposures affect the secrecy of keys minimally. We then lift our unidirectional variant to the fully bidirectional interaction setting in two meaningfully incremental steps. With this incremental approach, we reduce the significantly increased complexity of concurrent, bidirectional ratcheting in our corresponding security definitions and provably secure ratcheting constructions. These

new strongly secure, almost practical constructions—besides confirming satisfiability of our notions—reveal and illustrate an interesting gap in the cryptographic hardness of ratcheting: while our unidirectional scheme can be implemented with standard building blocks, we introduce key-updatable public key encryption to achieve security in the bidirectional setting.

This gap is further analyzed in Chapter 4, where we identify clear conditions under which key-updatable public key encryption is indeed necessary to realize ratcheting. Interestingly, this gap is not anchored between unidirectional and bidirectional ratcheting but already occurs in the unidirectional setting depending on the permitted adversarial power. With this result, we underline the importance of the utilized key-updating techniques for ratcheting, but also indicate how relying on them can be bypassed in order to achieve better performance. In the course of proving this relation, we develop a security definition of unidirectional ratcheting that also takes attacks against protocol executions' randomness into account. As we thereby follow our puristic definitional approach, a second core outcome in this chapter is our more practical and meaningfully stronger *natural* security notion of ratcheting.

In Chapter 5, we turn to the group setting in ratcheting and analyze its communication costs in case multiple group members send concurrently. Although concurrency has been extensively discussed in this context, specifically with respect to the current IETF Messaging Layer Security (MLS) initiative[1], none of the previous constructions was able to provide satisfying solutions. Again, we break away from the construction-perspective that, except for our work, is prevalent in the literature on group ratcheting, and instead consider this primitive from a theoretic point of view. To analyze the minimal necessary communication overhead that is required for achieving security after adversarial state exposures under concurrent sending, we adopt a proof technique that has so far only been used once for a similar pur-

[1]For example here: https://mailarchive.ietf.org/arch/msg/mls/oSArWtEqBzF1s5BpXGX11jaZrFI/

pose [MP04]: We develop a symbolic execution model that captures group ratcheting generically insofar that group ratcheting constructions are, by the model, meaningfully and practically limited in their use of cryptographic building blocks. In this model, we are able to prove that a communication overhead linear in the number of concurrently sending group members is inevitable. To complement this lower bound, we provide a simple, provably secure group ratcheting construction that has an almost tightly matching upper bound in communication complexity. These results may influence the design of the current or future standardized schemes in the MLS initiative. Furthermore, our lower bound manifests a theoretic limit in the performance of group ratcheting. Finally, our proof technique emphasizes the usefulness of, and may renew the interest in, symbolic modeling for the purpose of analyzing (communication) complexity limits.

Our systematizing view on according security models in Chapter 6 illustrates that the research on group ratcheting and, more generally, on group key exchange is indeed primarily construction-driven. One of our main motivations to systematize game-based security models of group key exchange in the literature is our conviction that the progress of research on a cryptographic primitive is significantly influenced by the clarity of its abstract conception. We review established syntax, correctness, and security definitions to clarify their suitability with respect to how they cover and restrict reality, and which (undesirable) implications they entail. Thereby, we also integrate practical requirements that arise from modern group ratcheting research. We discuss and compare contained solution strategies and label desirable approaches. Since no considered security model fulfills all characteristics that we regard desirable, and in order to show that these characteristics are fully compatible and achievable, we conclude this chapter with a corresponding simple and generic security model for group key exchange. With these results, we aim to support the emerging research on group ratcheting by proposing a clearer and more general concept of it.

7.2 Statefulness

A common property of all cryptographic primitives in this thesis is that they make use of a local modifiable state that stores user secrets. The way that this state is exploited to reach better security guarantees is relatively new in cryptographic literature.

Our Contributions With our definitions and constructions from chapters 3 and 4, we are the first to determine the theoretical optimum of security (here: secrecy of keys) that can be achieved by exploiting statefulness in two-party ratcheting. Similarly, with our symbolic analysis in Chapter 5, we are the first to determine the minimal communication complexity achievable in concurrent group ratcheting due to exploitation of statefulness. Finally, our equivalence result from Chapter 4 reduces statefulness of ratcheting to the simpler and more fundamental concept of public key encryption with updatable key pairs. These contributions illustrate that exploiting local user states is a very powerful and yet relatively sparsely explored tool in cryptography—simply enabled by adding to the syntax of algorithms an input and output parameter (see Section 7.3).

Open Questions and Future Work The main security goal that we consider in this thesis, and that is also primarily achieved by the regarded cryptographic primitives, is confidentiality. Exploiting statefulness is potentially also supportive for other security goals such as authentication, or privacy goals such as anonymity. We believe that continuing our steps in the analysis of stateful cryptography by including these goals is promising. An orthogonal goal, enabled by the exploitation of a user state, could be performance. Note that introducing a user state is not necessarily accompanied by strengthened adversaries. Hence, instead of increasing adversarial power and aiming for optimal security under introduced statefulness, constructions could utilize their state for better efficiency under steady security requirements.

7.3 Defining Syntax, Correctness, and Security

In our security notions, we attach great importance to methodically and carefully defining all components: syntax, correctness, and security. Especially in emerging research topics, like (group) ratcheting, we think that systematic considerations of the abstract concepts are vital to maintaining comprehensibility and comparability from the beginning. However, with our results from Chapter 6, we illustrate that these modeling definitions, constituting the system's core and theoretic hub for a primitive in cryptography, often only play a secondary role, even in established literature.

Our Contributions With our definitions from chapters 3, 4, and 6, we are the first to introduce simple, generic, powerful, and clear syntax notions, light correctness requirements, and natural security definitions that minimally restrict adversaries for the respective primitives. It is clear that methodical approaches are not superior per se. Note, for example, that some of our constructions, for achieving the required strong security, are relatively impractical. However, we believe that methodical analyses are necessary to explore the possibilities and limits of a cryptographic primitive, and expose internal components with their respective relations and properties. With our strategy, we are, for example, able to unveil interesting relations between ratcheting and key-updatable public key encryption in Chapter 4. A core contribution regarding systematic modeling is, of course, our systematizing literature review in Chapter 6 in which we examine generality, usefulness, and comparability of existing syntax, correctness, and security notions, and provide a precise, significantly simpler, and more generic model.

Open Questions and Future Work Our contributions are only limited to specific primitives, but our results may indicate that syntax specifications, when defining security, are far more influential than

currently understood. However, most syntax notions seemingly base on arbitrary human decisions. Equally, many security definitions (especially in key exchange) are based on human intuition.

Although defining correctness and syntax seems to be acknowledged as an inherently ambiguous process, a clearer understanding of naturally existing relations between syntax and security is desirable from our perspective. For example, we believe that a fundamental analysis of power induced by permitted inputs and outputs of algorithms (e.g., associated-data inputs, permanently accessible, modifiable state, static asymmetric public inputs, static asymmetric and symmetric secret inputs, etc.) is due. A first step towards this analysis could be a systematization of syntax notions and an exploration of general semantics that are already intuitively and, hence, implicitly included in existing cryptographic syntax definitions. For example, the role of associated data seems abstractly clear; it seems, however, less understood whether associated data should be treated as public or secret information (as for nonces [BNT19]), whether it is a symmetric input for all involved users or only input once and then derivable for all remaining users (cf. secret sharing [BDR20a]), etc. Developing natural approaches of defining syntax, and deriving correctness and security from it, similar to the *security up to correctness* formalization by Rogaway and Zhang [RZ18], could extend this step in an interesting direction for future research.

7.4 Continuous State Updates

The primary construction technique for satisfying security requirements in this thesis is updating the key material in local user states. This technique is essentially inspired by its integration in modern messengers, specifically in Signal's Double Ratchet Algorithm [PM16].

Our Contributions Our construction of unidirectional ratcheting from Chapter 3 is the first that generalizes utilized asymmetric cryptographic building blocks. While all previous works from practice and

academia propose ad-hoc ratcheting designs based on specific Diffie–Hellman groups, we expose the actually required properties by using generic key encapsulation mechanisms. With our constructions of sesqui- and bidirectional ratcheting from Chapter 3, we are the first to introduce key-updatable key encapsulation mechanisms as an entirely new update technique to achieve strong forward-secrecy for asymmetric key material. Abstractly, it can be seen as the asymmetric counterpart of symmetric key updates via hash-chains. By proving that these key-updatable techniques are indeed necessary to build strongly secure ratcheting in Chapter 4, we underline their importance. In our upper bound construction for group ratcheting from Chapter 5, we are the first to show how concurrently initiated state updates can be used to effectively recover from exposed secrets. For this we employ techniques from broadcast encryption.

Open Questions and Future Work So far, our strongly secure asymmetric key-update technique has only been implemented generically from (inefficient) hierarchical identity-based encryption (HIBE). As discussed in Chapter 4, it is, however, evident that weaker assumptions are sufficient. Finding a more efficient instantiation for key-updatable key encapsulation mechanisms remains an open problem. In a recently proposed optimally secure group ratcheting construction [ACJM20] key updates are realized via HIBE directly. Hence, analyzing corresponding hardness relations in the group setting remains an open problem as well.

Our security definition of group ratcheting in Chapter 5 requires recovery from state exposures only with a delay by one-round. We justify this relaxation in recovery speed with otherwise seemingly necessary use of multi-party non-interactive key exchange. An interesting open problem is to prove this necessity. On the construction side, we developed an unpublished, more complex but also slightly more efficient concurrent group ratcheting design from black-box broadcast encryption. Exploring whether recent successes in optimally efficient broadcast encryption [AY20] are furthermore applicable to achieve

better performance also remains an interesting question for future work.

7.5 Asynchronicity

Providing functionality under minimal reliability guarantees from the environment is a key feature of practical messaging applications. The Signal protocol is, for example, equipped with very practical techniques to handle asynchronous communication. Generally, it is also necessary for the ratcheting component in messaging apps that they can handle asynchronous interaction between the session participants.

Our Contributions We are the first to formally model asynchronous interaction in ratcheting with our definitions from Chapter 3. With these definitions and our accordingly secure constructions, we investigate the tension between optimal security, asynchronous communication, and performance. For example, we justify the use of key-updatable techniques in our bidirectionally interactive constructions with an example attack that exploits permitted concurrency: the adversary lets Bob recover from an exposure and simultaneously impersonates Alice undetectably towards Bob, which is required to remain harmless. Concurrent sending in groups is the focus of our analysis in Chapter 5, with which we are the first to solve concurrent state recovery in groups and determine necessary and sufficient communication costs induced by it. Nevertheless, our communication model in Chapter 5 still relies on a weak form of synchronicity: a reliable round schedule (i.e., synchronized clocks).

Open Questions and Future Work Lifting our results from Chapter 5 to the fully asynchronous setting remains an open problem. Another interesting question is raised by the functionality guarantees of Signal: This protocol can recover from state exposures even under out of order receipts and message loss. The security analysis by Alwen et al. [ACD19] took account of this, but neither an analysis of potential

improvements in performance or security, nor an extension to group conversation exists.

7.6 Two Perspectives on Problems

We think that one core contribution of our work is that we consider all research questions from two perspectives: Finding a concrete solution that satisfies the requirements of a raised problem, and analyzing the respective limits of satisfiability in general.

With our unidirectional ratcheting construction from Chapter 3, we demonstrate the sufficiency of generic key encapsulation mechanisms; The necessity of key encapsulation mechanisms for this notion of ratcheting is trivially evident. Our analysis in Chapter 4 is exclusively devoted to understanding the necessary and sufficient conditions for realizing strongly secure ratcheting from key-updatable key encapsulation mechanisms. Also with our consideration of concurrent group ratcheting in Chapter 5 we prove almost tight lower and upper bounds of communication complexity. In a broader sense even our systematization from Chapter 6 includes the discovery of desirable characteristics of group key exchange models for which we show with our new model that they are indeed satisfiable. Finally, even in work that has been conducted independent of this thesis, we follow this approach: for our design of authenticated encryption combiners [PR20], we show that any combiner, invoking every algorithm of each underlying scheme at most once, is insecure; our proposed secure combiner invokes the underlying schemes' algorithms exactly once plus one extra invocation of one of these algorithms.

We believe that this approach is essential for justifying a proposed solution, showing that it does not only meet the necessary requirements, but it meets these requirements with little (potentially even minimal) overhead.

7.7 Impact

Many of our results are foremost influential on the academic research itself as they constitute clear frameworks, point out theoretic possibilities and limits, as well as conceptualize relations. Their applicability on practice is yet not entirely foreseeable, but a better understanding of abstract concepts can be an important catalyst for a research field—eventually enabling practical solutions.

Some of our results directly contribute to solutions for questions from practice. Most importantly, our concurrent group ratcheting scheme is immediately applicable to the standardization effort of the MLS initiative.

In this chapter we point out various open questions and promising research directions that can extend our results. We think that revealing open problems and indicating room for generalization may be one of our main contributions and can significantly impact future research.

Bibliography

[ACC+19a] Joël Alwen, Margarita Capretto, Miguel Cueto, Chethan Kamath, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. Keep the dirt: Tainted treekem, an efficient and provably secure continuous group key agreement protocol. Cryptology ePrint Archive, Report 2019/1489, 2019. https://eprint.iacr.org/2019/1489.

[ACC+19b] Joël Alwen, Margarita Capretto, Miguel Cueto, Chethan Kamath, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. Keep the dirt: Tainted treekem, an efficient and provably secure continuous group key agreement protocol. Cryptology ePrint Archive, Report 2019/1489, 2019. Fixed version of [ACC+19a] for detailed comparison: https://eprint.iacr.org/2019/1489 downloaded 2020-02-13.

[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.

[ACDT19] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. Cryptology ePrint Archive, Report 2019/1189, 2019. Fixed version of [ACDT20] for detailed comparison: https://eprint.iacr.org/2019/1189 downloaded 2020-02-13.

[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.

[ACJM20] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In *Theory of Cryptography - 18th International Conference, TCC 2020, November 15-19, 2020, Proceedings*, Lecture Notes in Computer Science. Springer, 2020.

[AY20] Shweta Agrawal and Shota Yamada. Optimal broadcast encryption from pairings and LWE. In Anne Canteaut and Yuval Ishai, editors, *Advances in Cryptology – EUROCRYPT 2020, Part I*, volume 12105 of *Lecture Notes in Computer Science*, pages 13–43, Zagreb, Croatia, May 10–14, 2020. Springer, Heidelberg, Germany.

[BBM+20a] R. Barnes, B. Beurdouche, J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. The messaging layer security (mls) protocol. Technical report, 2020. https://datatracker.ietf.org/doc/draft-ietf-mls-protocol/.

[BBM+20b] R. Barnes, B. Beurdouche, J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. The messaging layer security (MLS) protocol draft-ietf-mls-protocol-09. Internet-draft, September 2020.

[BBP04] Mihir Bellare, Alexandra Boldyreva, and Adriana Palacio. An uninstantiable random-oracle-model scheme for a hybrid-encryption problem. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Com-*

puter Science, pages 171–188, Interlaken, Switzerland, May 2–6, 2004. Springer, Heidelberg, Germany.

[BCP01] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. Provably authenticated group Diffie-Hellman key exchange – the dynamic case. In Colin Boyd, editor, *Advances in Cryptology – ASIACRYPT 2001*, volume 2248 of *Lecture Notes in Computer Science*, pages 290–309, Gold Coast, Australia, December 9–13, 2001. Springer, Heidelberg, Germany.

[BCP02a] 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.

[BCP02b] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. Group Diffie-Hellman key exchange secure against dictionary attacks. In Yuliang Zheng, editor, *Advances in Cryptology – ASIACRYPT 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 497–514, Queenstown, New Zealand, December 1–5, 2002. Springer, Heidelberg, Germany.

[BCPQ01] Emmanuel Bresson, Olivier Chevassut, David Pointcheval, and Jean-Jacques Quisquater. Provably authenticated group Diffie-Hellman key exchange. In Michael K. Reiter and Pierangela Samarati, editors, *ACM CCS 2001: 8th Conference on Computer and Communications Security*, pages 255–264, Philadelphia, PA, USA, November 5–8, 2001. ACM Press.

[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.

[BD05] Mike Burmester and Yvo Desmedt. A secure and scalable group key exchange system. *Inf. Process. Lett.*, 94(3):137–143, 2005.

[BDR20a] Mihir Bellare, Wei Dai, and Phillip Rogaway. Reimagining secret sharing: Creating a safer and more versatile primitive by adding authenticity, correcting errors, and reducing randomness requirements. *Proc. Priv. Enhancing Technol.*, 2020(4):461–490, 2020.

[BDR20b] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. In *Theory of Cryptography - 18th International Conference, TCC 2020, Virtual, November 15-19, 2020, Proceedings*, Lecture Notes in Computer Science. Springer, 2020.

[BDR20c] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. Cryptology ePrint Archive, Report 2020/1171, 2020. https://eprint.iacr.org/2020/1171.

[BFWW11] Christina Brzuska, Marc Fischlin, Bogdan Warinschi, and Stephen C. Williams. Composability of Bellare-Rogaway key exchange protocols. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011: 18th Conference on Computer and Communications Security*, pages 51–62, Chicago, Illinois, USA, October 17–21, 2011. ACM Press.

[BGB04] Nikita Borisov, Ian Goldberg, and Eric A. Brewer. Off-the-record communication, or, why not to use PGP. In

Vijay Atluri, Paul F. Syverson, and Sabrina De Capitani di Vimercati, editors, *Proceedings of the 2004 ACM WPES 2004, Washington, DC, USA, October 28, 2004*, pages 77–84. ACM, 2004.

[BGK+18] Dan Boneh, Darren Glass, Daniel Krashen, Kristin Lauter, Shahed Sharif, Alice Silverberg, Mehdi Tibouchi, and Mark Zhandry. Multiparty non-interactive key exchange and more from isogenies on elliptic curves. Cryptology ePrint Archive, Report 2018/665, 2018. https://eprint.iacr.org/2018/665.

[BHK15] Mihir Bellare, Dennis Hofheinz, and Eike Kiltz. Subtleties in the definition of IND-CCA: When and how should challenge decryption be disallowed? *Journal of Cryptology*, 28(1):29–48, January 2015.

[BL15] Mihir Bellare and Anna Lysyanskaya. Symmetric and dual PRFs from standard assumptions: A generic validation of an HMAC assumption. Cryptology ePrint Archive, Report 2015/1198, 2015. http://eprint.iacr.org/2015/1198.

[BNT19] Mihir Bellare, Ruth Ng, and Björn Tackmann. Nonces are noticed: AEAD revisited. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part I*, volume 11692 of *Lecture Notes in Computer Science*, pages 235–265, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany.

[Box76] George EP Box. Science and statistics. *Journal of the American Statistical Association*, 71(356):791–799, 1976.

[BR94] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *Advances in Cryptology – CRYPTO'93*, volume 773

of *Lecture Notes in Computer Science*, pages 232–249, Santa Barbara, CA, USA, August 22–26, 1994. Springer, Heidelberg, Germany.

[BRV20a] Fatih Balli, Paul Rösler, and Serge Vaudenay. Determining the core primitive for optimally secure ratcheting. In *Advances in Cryptology - ASIACRYPT 2020 - 26th International Conference on the Theory and Application of Cryptology and Information Security, Virtual, December 7-11, 2020, Proceedings*, Lecture Notes in Computer Science, 2020.

[BRV20b] Fatih Balli, Paul Rösler, and Serge Vaudenay. Determining the core primitive for optimally secure ratcheting. Cryptology ePrint Archive, Report 2020/148, 2020. https://eprint.iacr.org/2020/148.

[BS02] Dan Boneh and Alice Silverberg. Applications of multilinear forms to cryptography. Cryptology ePrint Archive, Report 2002/080, 2002. http://eprint.iacr.org/2002/080.

[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.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd Annual Symposium on Foundations of Computer Science*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

[CCD+17] Katriel Cohn-Gordon, Cas J. F. Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. In *2017 IEEE EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 451–466. IEEE, 2017.

[CCG16] Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. On post-compromise security. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 164–178. IEEE Computer Society, 2016.

[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.

[CCG+19] Katriel Cohn-Gordon, Cas Cremers, Kristian Gjøsteen, Håkon Jacobsen, and Tibor Jager. Highly efficient key exchange protocols with optimal tightness. In Alexandra Boldyreva and Daniele Micciancio, editors, *Advances in Cryptology – CRYPTO 2019, Part III*, volume 11694 of *Lecture Notes in Computer Science*, pages 767–797, Santa Barbara, CA, USA, August 18–22, 2019. Springer, Heidelberg, Germany.

[CDV19] Andrea Caforio, F Betül Durak, and Serge Vaudenay. On-demand ratcheting with security awareness. Cryptology ePrint Archive, Report 2019/965, 2019. https://eprint.iacr.org/2019/965.

[CGH98a] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited. Cryptology ePrint

Archive, Report 1998/011, 1998. http://eprint.iacr.org/1998/011.

[CGH98b] Ran Canetti, Oded Goldreich, and Shai Halevi. The random oracle methodology, revisited (preliminary version). In *30th Annual ACM Symposium on Theory of Computing*, pages 209–218, Dallas, TX, USA, May 23–26, 1998. ACM Press.

[CHK04] Ran Canetti, Shai Halevi, and Jonathan Katz. Chosen-ciphertext security from identity-based encryption. In Christian Cachin and Jan Camenisch, editors, *Advances in Cryptology – EUROCRYPT 2004*, volume 3027 of *Lecture Notes in Computer Science*, pages 207–222, Interlaken, Switzerland, May 2–6, 2004. Springer, Heidelberg, Germany.

[CHK19] Cas Cremers, Britta Hale, and Konrad Kohbrok. Efficient post-compromise security beyond one group. Cryptology ePrint Archive, Report 2019/477, 2019. https://eprint.iacr.org/2019/477.

[CNE+14] Stephen Checkoway, Ruben Niederhagen, Adam Everspaugh, Matthew Green, Tanja Lange, Thomas Ristenpart, Daniel J. Bernstein, Jake Maskiewicz, Hovav Shacham, and Matthew Fredrikson. On the practical exploitability of dual EC in TLS implementations. In Kevin Fu and Jaeyeon Jung, editors, *USENIX Security 2014: 23rd USENIX Security Symposium*, pages 319–335, San Diego, CA, USA, August 20–22, 2014. USENIX Association.

[DFGS15] Benjamin Dowling, Marc Fischlin, Felix Günther, and Douglas Stebila. A cryptographic analysis of the TLS 1.3 handshake protocol candidates. In Indrajit Ray, Ninghui Li, and Christopher Kruegel, editors, *ACM CCS 2015:*

22nd Conference on Computer and Communications Security, pages 1197–1210, Denver, CO, USA, October 12–16, 2015. ACM Press.

[DJSS18] David Derler, Tibor Jager, Daniel Slamanig, and Christoph Striecks. Bloom filter encryption and applications to efficient forward-secret 0-RTT key exchange. In Jesper Buus Nielsen and Vincent Rijmen, editors, *Advances in Cryptology – EUROCRYPT 2018, Part III*, volume 10822 of *Lecture Notes in Computer Science*, pages 425–455, Tel Aviv, Israel, April 29 – May 3, 2018. Springer, Heidelberg, Germany.

[DRS20] Benjamin Dowling, Paul Rösler, and Jörg Schwenk. Flexible authenticated and confidential channel establishment (fACCE): Analyzing the noise protocol framework. In Aggelos Kiayias, Markulf Kohlweiss, Petros Wallden, and Vassilis Zikas, editors, *PKC 2020: 23rd International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 12110 of *Lecture Notes in Computer Science*, pages 341–373, Edinburgh, UK, May 4–7, 2020. Springer, Heidelberg, Germany.

[DV18] F. Betül Durak and Serge Vaudenay. Bidirectional asynchronous ratcheted key agreement with linear complexity. Cryptology ePrint Archive, Report 2018/889, 2018. https://eprint.iacr.org/2018/889.

[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.

[DY81] Danny Dolev and Andrew Chi-Chih Yao. On the security of public key protocols (extended abstract). In *22nd Annual Symposium on Foundations of Computer Science*, pages 350–357, Nashville, TN, USA, October 28–30, 1981. IEEE Computer Society Press.

[EMP18] Patrick Eugster, Giorgia Azzurra Marson, and Bertram Poettering. A cryptographic look at multi-party channels. In *31st IEEE Computer Security Foundations Symposium, CSF 2018, Oxford, United Kingdom, July 9-12, 2018*, pages 31–45. IEEE Computer Society, 2018.

[FG14] Marc Fischlin and Felix Günther. Multi-stage key exchange and the case of Google's QUIC protocol. In Gail-Joon Ahn, Moti Yung, and Ninghui Li, editors, *ACM CCS 2014: 21st Conference on Computer and Communications Security*, pages 1193–1204, Scottsdale, AZ, USA, November 3–7, 2014. ACM Press.

[FG17] Marc Fischlin and Felix Günther. Replay attacks on zero round-trip time: The case of the TLS 1.3 handshake candidates. In *2017 IEEE European Symposium on Security and Privacy, EuroS&P 2017, Paris, France, April 26-28, 2017*, pages 60–75. IEEE, 2017.

[FN94] Amos Fiat and Moni Naor. Broadcast encryption. In Douglas R. Stinson, editor, *Advances in Cryptology – CRYPTO'93*, volume 773 of *Lecture Notes in Computer Science*, pages 480–491, Santa Barbara, CA, USA, August 22–26, 1994. Springer, Heidelberg, Germany.

[FS87] Amos Fiat and Adi Shamir. How to prove yourself: Practical solutions to identification and signature problems. In Andrew M. Odlyzko, editor, *Advances in Cryptology – CRYPTO'86*, volume 263 of *Lecture Notes in Computer Science*, pages 186–194, Santa Barbara, CA, USA, August 1987. Springer, Heidelberg, Germany.

[GBG09] M. Choudary Gorantla, Colin Boyd, and Juan Manuel González Nieto. Modeling key compromise impersonation attacks on group key exchange protocols. In Stanislaw Jarecki and Gene Tsudik, editors, *PKC 2009: 12th International Conference on Theory and Practice of Public Key Cryptography*, volume 5443 of *Lecture Notes in Computer Science*, pages 105–123, Irvine, CA, USA, March 18–20, 2009. Springer, Heidelberg, Germany.

[GHJL17] Felix Günther, Britta Hale, Tibor Jager, and Sebastian Lauer. 0-RTT key exchange with full forward secrecy. In Jean-Sébastien Coron and Jesper Buus Nielsen, editors, *Advances in Cryptology – EUROCRYPT 2017, Part III*, volume 10212 of *Lecture Notes in Computer Science*, pages 519–548, Paris, France, April 30 – May 4, 2017. Springer, Heidelberg, Germany.

[GHP18] Federico Giacon, Felix Heuer, and Bertram Poettering. KEM combiners. In Michel Abdalla and Ricardo Dahab, editors, *PKC 2018: 21st International Conference on Theory and Practice of Public Key Cryptography, Part I*, volume 10769 of *Lecture Notes in Computer Science*, pages 190–218, Rio de Janeiro, Brazil, March 25–29, 2018. Springer, Heidelberg, Germany.

[GMR+16] Martin Grothe, Christian Mainka, Paul Rösler, Johanna Jupke, Jan Kaiser, and Jörg Schwenk. Your cloud in my company: Modern rights management services revisited. In *11th International Conference on Availability, Reliability and Security, ARES 2016, Salzburg, Austria, August 31 - September 2, 2016*, pages 217–222. IEEE Computer Society, 2016.

[GMRS16] Martin Grothe, Christian Mainka, Paul Rösler, and Jörg Schwenk. How to break Microsoft rights management services. In Natalie Silvanovich and Patrick Traynor, ed-

itors, *10th USENIX Workshop on Offensive Technologies, WOOT 16, Austin, TX, USA, August 8-9, 2016*. USENIX Association, 2016.

[GS02] Craig Gentry and Alice Silverberg. Hierarchical ID-based cryptography. In Yuliang Zheng, editor, *Advances in Cryptology – ASIACRYPT 2002*, volume 2501 of *Lecture Notes in Computer Science*, pages 548–566, Queenstown, New Zealand, December 1–5, 2002. Springer, Heidelberg, Germany.

[GT00] Rosario Gennaro and Luca Trevisan. Lower bounds on the efficiency of generic cryptographic constructions. In *41st Annual Symposium on Foundations of Computer Science*, pages 305–313, Redondo Beach, CA, USA, November 12–14, 2000. IEEE Computer Society Press.

[HDWH12] Nadia Heninger, Zakir Durumeric, Eric Wustrow, and J. Alex Halderman. Mining your ps and qs: Detection of widespread weak keys in network devices. In Tadayoshi Kohno, editor, *USENIX Security 2012: 21st USENIX Security Symposium*, pages 205–220, Bellevue, WA, USA, August 8–10, 2012. USENIX Association.

[HS02] Dani Halevy and Adi Shamir. The LSD broadcast encryption scheme. In Moti Yung, editor, *Advances in Cryptology – CRYPTO 2002*, volume 2442 of *Lecture Notes in Computer Science*, pages 47–60, Santa Barbara, CA, USA, August 18–22, 2002. Springer, Heidelberg, Germany.

[IR89] Russell Impagliazzo and Steven Rudich. Limits on the provable consequences of one-way permutations. In *21st Annual ACM Symposium on Theory of Computing*, pages 44–61, Seattle, WA, USA, May 15–17, 1989. ACM Press.

[ITW82] Ingemar Ingemarsson, Donald Tang, and C Wong. A conference key distribution system. *IEEE Transactions on Information Theory*, 28(5):714–720, 1982.

[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.

[JS18a] 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.

[JS18b] Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. Cryptology ePrint Archive, Report 2018/553, 2018. https://eprint.iacr.org/2018/553.

[KLL04] Hyun-Jeong Kim, Su-Mi Lee, and Dong Hoon Lee. Constant-round authenticated group key exchange for dynamic groups. In Pil Joong Lee, editor, *Advances in Cryptology – ASIACRYPT 2004*, volume 3329 of *Lecture Notes in Computer Science*, pages 245–259, Jeju Island, Korea, December 5–9, 2004. Springer, Heidelberg, Germany.

[KS05] Jonathan Katz and Ji Sun Shin. Modeling insider attacks on group key-exchange protocols. In Vijayalakshmi Atluri, Catherine Meadows, and Ari Juels, editors, *ACM*

CCS 2005: 12th Conference on Computer and Communications Security, pages 180–189, Alexandria, Virginia, USA, November 7–11, 2005. ACM Press.

[KY03] Jonathan Katz and Moti Yung. Scalable protocols for authenticated group key exchange. In Dan Boneh, editor, *Advances in Cryptology – CRYPTO 2003*, volume 2729 of *Lecture Notes in Computer Science*, pages 110–125, Santa Barbara, CA, USA, August 17–21, 2003. Springer, Heidelberg, Germany.

[Lan16] Adam Langley. Source code of Pond. https://github.com/agl/pond, 05 2016.

[LS17] Yong Li and Sven Schäge. No-match attacks and robust partnering definitions: Defining trivial attacks for security protocols is not trivial. In Bhavani M. Thuraisingham, David Evans, Tal Malkin, and Dongyan Xu, editors, *ACM CCS 2017: 24th Conference on Computer and Communications Security*, pages 1343–1360, Dallas, TX, USA, October 31 – November 2, 2017. ACM Press.

[MBP+19a] Jens Müller, Marcus Brinkmann, Damian Poddebniak, Hanno Böck, Sebastian Schinzel, Juraj Somorovsky, and Jörg Schwenk. "Johnny, you are fired!" - Spoofing OpenPGP and S/MIME signatures in emails. In Nadia Heninger and Patrick Traynor, editors, *USENIX Security 2019: 28th USENIX Security Symposium*, pages 1011–1028, Santa Clara, CA, USA, August 14–16, 2019. USENIX Association.

[MBP+19b] Jens Müller, Marcus Brinkmann, Damian Poddebniak, Sebastian Schinzel, and Jörg Schwenk. Re: What's up johnny? - Covert content attacks on email end-to-end encryption. In Robert H. Deng, Valérie Gauthier-Umaña, Martín Ochoa, and Moti Yung, editors, *ACNS 19: 17th International Conference on Applied Cryptography and*

Network Security, volume 11464 of *Lecture Notes in Computer Science*, pages 24–42, Bogota, Colombia, June 5–7, 2019. Springer, Heidelberg, Germany.

[MBP+20] Jens Müller, Marcus Brinkmann, Damian Poddebniak, Sebastian Schinzel, and Jörg Schwenk. Mailto: Me your secrets. on bugs and features in email end-to-end encryption. In *8th IEEE Conference on Communications and Network Security, CNS 2020, Avignon, France, June 29 - July 1, 2020*, pages 1–9. IEEE, 2020.

[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.

[MP17] Giorgia Azzurra Marson and Bertram Poettering. Security notions for bidirectional channels. *IACR Transactions on Symmetric Cryptology*, 2017(1):405–426, 2017.

[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.

[MRY04] Philip D. MacKenzie, Michael K. Reiter, and Ke Yang. Alternatives to non-malleability: Definitions, constructions, and applications (extended abstract). In Moni Naor, editor, *TCC 2004: 1st Theory of Cryptography Conference*, volume 2951 of *Lecture Notes in Computer*

Science, pages 171–190, Cambridge, MA, USA, February 19–21, 2004. Springer, Heidelberg, Germany.

[NNL01] Dalit Naor, Moni Naor, and Jeffery Lotspiech. Revocation and tracing schemes for stateless receivers. In Joe Kilian, editor, *Advances in Cryptology – CRYPTO 2001*, volume 2139 of *Lecture Notes in Computer Science*, pages 41–62, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany.

[NY90] Moni Naor and Moti Yung. Public-key cryptosystems provably secure against chosen ciphertext attacks. In *22nd Annual ACM Symposium on Theory of Computing*, pages 427–437, Baltimore, MD, USA, May 14–16, 1990. ACM Press.

[OTR16] Off-the-Record Messaging. http://otr.cypherpunks.ca, 2016.

[PDM+18] Damian Poddebniak, Christian Dresen, Jens Müller, Fabian Ising, Sebastian Schinzel, Simon Friedberger, Juraj Somorovsky, and Jörg Schwenk. Efail: Breaking S/MIME and OpenPGP email encryption using exfiltration channels. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018: 27th USENIX Security Symposium*, pages 549–566, Baltimore, MD, USA, August 15–17, 2018. USENIX Association.

[Per17] Trevor Perrin. The noise protocol framework. http://noiseprotocol.org/noise.html, 2017. Revision 33.

[PM16] Trevor Perrin and Moxie Marlinspike. The double ratchet algorithm. https://whispersystems.org/docs/specifications/doubleratchet/doubleratchet.pdf, 11 2016.

[PR18a] Bertram Poettering and Paul Rösler. Asynchronous ratcheted key exchange. Cryptology ePrint Archive, Report 2018/296, 2018. https://eprint.iacr.org/2018/296.

[PR18b] 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.

[PR20] Bertram Poettering and Paul Rösler. Combiners for AEAD. *IACR Transactions on Symmetric Cryptology*, 2020(1):121–143, 2020.

[PRSS21] Bertram Poettering, Paul Rösler, Jörg Schwenk, and Douglas Stebila. SoK: Game-based security models for group key exchange, 2021.

[PSS+] Damian Poddebniak, Juraj Somorovsky, Sebastian Schinzel, Manfred Lochter, and Paul Rösler. Attacking deterministic signature schemes using fault attacks. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 338–352. IEEE.

[RMS18] Paul Rösler, Christian Mainka, and Jörg Schwenk. More is less: On the end-to-end security of group chats in Signal, WhatsApp, and Threema. In *2018 IEEE European Symposium on Security and Privacy, EuroS&P 2018, London, United Kingdom, April 24-26, 2018*, pages 415–429. IEEE, 2018.

[Rog02] Phillip Rogaway. Authenticated-encryption with associated-data. In Vijayalakshmi Atluri, editor, *ACM*

CCS 2002: 9th Conference on Computer and Communications Security, pages 98–107, Washington, DC, USA, November 18–22, 2002. ACM Press.

[Rös15] Paul Rösler. Architektur- und sicherheitsanalyse von Tresorit und Tresorit DRM. Bachelor's thesis, Ruhr University Bochum, 9 2015.

[Rös18] Paul Rösler. On the end-to-end security of group chats in instant messaging protocols. Master's thesis, Ruhr University Bochum, 12 2018.

[Rös19] Paul Rösler. Why receipt notifications increase security in signal. https://web-in-security.blogspot.com/2019/05/acks-for-security.html, 2019.

[RS92] Charles Rackoff and Daniel R. Simon. Non-interactive zero-knowledge proof of knowledge and chosen ciphertext attack. In Joan Feigenbaum, editor, *Advances in Cryptology – CRYPTO'91*, volume 576 of *Lecture Notes in Computer Science*, pages 433–444, Santa Barbara, CA, USA, August 11–15, 1992. Springer, Heidelberg, Germany.

[RS09] Eric Rescorla and Margaret Salter. Extended random values for tls. Technical report, 2009.

[RZ18] Phillip Rogaway and Yusi Zhang. Simplifying game-based definitions - indistinguishability up to correctness and its application to stateful AE. In Hovav Shacham and Alexandra Boldyreva, editors, *Advances in Cryptology – CRYPTO 2018, Part II*, volume 10992 of *Lecture Notes in Computer Science*, pages 3–32, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[Sho99] Victor Shoup. On formal models for secure key exchange. Technical Report RZ 3120, IBM, 1999.

[Wei19] Matthew Weidner. *Group messaging for secure asynchronous collaboration*. PhD thesis, MPhil Dissertation, 2019. Advisors: A. Beresford and M. Kleppmann, 2019, 2019.

[YKLH18] Zheng Yang, Mohsin Khan, Wanping Liu, and Jun He. On security analysis of generic dynamic authenticated group key exchange. In *Secure IT Systems - 23rd Nordic Conference, NordSec 2018, Oslo, Norway, November 28-30, 2018, Proceedings*, pages 121–137, 2018.

[YRS+09] Scott Yilek, Eric Rescorla, Hovav Shacham, Brandon Enright, and Stefan Savage. When private keys are public: results from the 2008 debian openssl vulnerability. In *Proceedings of the 9th ACM SIGCOMM Internet Measurement Conference, IMC 2009, Chicago, Illinois, USA, November 4-6, 2009*, pages 15–27, 2009.

[ZJC11] P. Zimmermann, A. Johnston, and J. Callas. ZRTP: Media path key agreement for unicast secure RTP. RFC 6189, RFC Editor, April 2011.