UC San Diego
UC San Diego Electronic Theses and Dissertations

Title
Secure Messaging: From Systems to Theory

Permalink
https://escholarship.org/uc/item/7kd1q23d

Author
Jaeger, Joseph Sumner

Publication Date
2019

Peer reviewed|Thesis/dissertation

eScholarship.org Powered by the California Digital Library
University of California

UNIVERSITY OF CALIFORNIA SAN DIEGO

Secure Messaging: From Systems to Theory

A dissertation submitted in partial satisfaction of the
requirements for the degree of Doctor of Philosophy

in

Computer Science

by

Joseph Jaeger

Committee in charge:

Professor Mihir Bellare, Chair
Professor Massimo Franceschetti
Professor Daniele Micciancio
Professor Stefan Savage
Professor Deian Stefan

2019

Copyright

Joseph Jaeger, 2019

All rights reserved.

The Dissertation of Joseph Jaeger is approved, and it is acceptable in quality and form for publication on microfilm and electronically:

____

____

____

____

____

Chair

University of California San Diego

2019

TABLE OF CONTENTS

Signature Page iii

Table of Contents iv

List of Figures vi

List of Tables xi

Acknowledgements xii

Vita xiv

Abstract of the Dissertation xv

Introduction 1

Chapter 1 Ratcheted Encryption and Key Exchange 7
1.1 Preliminaries 13
1.2 Oracle Diffie-Hellman with Exposures 16
1.3 Ratcheted key exchange 19
1.3.1 Definition of ratcheted key exchange 20
1.3.2 Security of ratcheted key exchange 23
1.3.3 Construction of a ratcheted key exchange scheme 28
1.3.4 Security proof for our ratcheted key exchange scheme 33
1.4 Ratcheted encryption 43
1.5 Oracle Diffie-Hellman with Exposures in ROM 51
1.5.1 ODHE reduction to SCDH in ROM 55
1.5.2 SCDHE reduction to SCDH with rewinding 62
1.6 Necessity of authenticaticating the update information 69
1.7 Acknowledgements 74

Chapter 2 Optimal Channel Security Against Fine-Grained State Compromise 75
2.1 Preliminaries 81
2.2 New asymmetric primitives 83
2.2.1 Key-updatable digital signature schemes 84
2.2.2 Key-updatable public-key encryption schemes 88
2.3 Bidirectional cryptographic channels 91
2.4 Security notion for channels 94
2.4.1 Channel interface game 95
2.4.2 Optimal security of a channel 99
2.4.3 Informal description of our security definition 103
2.5 Construction of a secure channel 105
2.5.1 Our construction 105

2.5.2 Security proof 113
2.6 Comparison to recent definitions 131
2.7 Security implications of correctness notions 134
2.8 Construction of key-updatable digital signatures 135
2.9 Construction of key-updatable public-key encryption 140
2.10 Acknowledgements 146

Chapter 3 Key Exchange for Messaging Apps 147
3.1 Notation and conventions 152
3.2 Messaging Security 152
3.3 Key exchange 156
3.3.1 Adversary Model 158
3.3.2 Stand-alone security of OKE 167
3.4 Generic Construction 174
3.5 Composition 177
3.6 Simulator Pseudocode 180
3.7 Formalization of Parameters 181
3.8 Composition proof 184
3.8.1 Proof of Theorem 15 189
3.9 Security of GKE 194
3.9.1 Proof of Theorem 17 200
3.9.2 More proof details (Claim 1) 215
3.9.3 More proof details (Claim 6) 219
3.10 Acknowledgements 230

Bibliography 232

LIST OF FIGURES

Figure 1.1. Games defining strong unforgeability of function family F under chosen message attack, and multi-user authenticated encryption security of SE. . . 14

Figure 1.2. Game defining Oracle Diffie-Hellman with Exposures assumption for $\mathbb{G}, \mathsf{H}$. 17

Figure 1.3. The interaction between ratcheted key exchange algorithms. 20

Figure 1.4. Game RKE-COR defining correctness of ratcheted key exchange scheme R, and game RE-COR defining correctness of ratcheted encryption scheme R. Oracles UP and RATREC are used in both games, whereas oracle ENC is only used in game RE-COR. 22

Figure 1.5. Games defining key indistinguishability of ratcheted key exchange scheme RKE, and authenticated encryption security of ratcheted encryption scheme RE. 27

Figure 1.6. Ratcheted key exchange scheme $\mathsf{RKE} = \mathsf{RATCHET\text{-}KE}[\mathbb{G}, \mathsf{F}, \mathsf{H}]$. 29

Figure 1.7. Attacks against insecure variants of $\mathsf{RKE} = \mathsf{RATCHET\text{-}KE}[\mathbb{G}, \mathsf{F}, \mathsf{H}]$. 29

Figure 1.8. Games $G_{0,j}, G^*_{0,j}, I_j$ for proof of Theorem 1. 36

Figure 1.9. Games G_1, G_2 for proof of Theorem 1. 37

Figure 1.10. Adversary $\mathcal{O}_1$ for proof of Theorem 1. 39

Figure 1.11. Adversary $\mathcal{F}$ for proof of Theorem 1. 40

Figure 1.12. Adversary $\mathcal{O}_2$ for proof of Theorem 1. 41

Figure 1.13. Ratcheted encryption scheme $\mathsf{RE} = \mathsf{RATCHET\text{-}ENC}[\mathsf{RKE}, \mathsf{SE}]$. 45

Figure 1.14. Games G_0, G_1 for proof of Theorem 2. 48

Figure 1.15. Adversary $\mathcal{D}$ for proof of Theorem 2. 49

Figure 1.16. Adversary $\mathcal{N}$ for proof of Theorem 2. 50

Figure 1.17. Game defining Oracle Diffie-Hellman with Exposures in ROM assumption for $\mathbb{G}, \mathsf{H}$. 52

Figure 1.18. Games defining Strong Computational Diffie-Hellman assumption in group $\mathbb{G}$, and Strong Computational Diffie-Hellman with Exposures assumption in group $\mathbb{G}$. 53

Figure 1.19. Games G_0, G_1 for proof of Lemma 5. 57

Figure 1.20. Adversary $\mathcal{B}$ for proof of Lemma 5. 59

Figure 1.21. Game defining SCDHE assumption in group $\mathbb{G}$. This non-black-box definition extends that of Fig. 1.18 to require that adversary $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$ halts after every oracle call. 64

Figure 1.22. Adversary $\mathcal{S}_u$ for proof of Lemma 7. 65

Figure 1.23. Games G_0, G_1 for proof of Lemma 7. 67

Figure 1.24. Game defining update unforgeability of ratcheted key exchange scheme RKE. 69

Figure 2.1. Game defining collision-resistance of function family H. 82

Figure 2.2. Games defining correctness of key-updatable digital signature scheme DS and correctness of key-updatable public-key encryption scheme PKE. 84

Figure 2.3. Game defining signature uniqueness of key-updatable digital signature scheme DS. 85

Figure 2.4. Games defining signature unforgeability under exposures of key-updatable digital signature scheme DS, and ciphertext indistinguishability under exposures of key-updatable public-key encryption scheme PKE. 86

Figure 2.5. A basic interaction between bidirectional cryptographic channel algorithms. 91

Figure 2.6. Games defining correctness of channel Ch. Lines labelled with the name of a game are included only in that game. 93

Figure 2.7. Game defining interface between adversary $\mathcal{D}$ and channel Ch. 95

Figure 2.8. Generic attacks against any channel Ch with interface INTER. 98

Figure 2.9. Game defining AEAC security of channel Ch. 100

Figure 2.10. Construction of channel $\mathsf{SCh} = \mathsf{SCH}[\mathsf{DS}, \mathsf{PKE}, \mathsf{H}]$ from function family H, key-updatable digital signature scheme DS, and key-updatable public-key encryption scheme PKE. 107

Figure 2.11. Attacks against variants of SCh. 109

Figure 2.12. Attacks against variants of SCh. 111

Figure 2.13. Games G_1, G_2, and G_3 for security proof. Lines commented with the names of games are only included in those games. Highlighting indicates new code. 116

Figure 2.14. Adversary $\mathcal{A}_{\mathsf{H}}$ against collision resistance of H. Highlighting indicates the changes from G_1, G_2. 119

Figure 2.15. Adversary $\mathcal{A}_{\mathsf{DS}}$ attacking DS. Highlighting indicates changes from G_2, G_3 121

Figure 2.16. Games G_4 and G_5 for security proof. Lines commented with the names of games are only included in those games. Highighted codes indicates changes from G_3. 125

Figure 2.17. Adversary $\mathcal{B}_{\mathsf{DS}}$ attacking DS. Highlighting indicates changes from G_4, G_5. 126

Figure 2.18. Adversary $\mathcal{A}_{\mathsf{PKE}}$ attacking PKE. Highlighting indicates changes from G_4, G_5. 128

Figure 2.19. Games defining correctness, uniqueness, and forward security of key-evolving digital signature scheme DS. 136

Figure 2.20. Key-updatable digital signature scheme $\mathsf{DS}_{\mathrm{KU}} = \mathsf{DS\text{-}CONS}[\mathsf{DS}_{\mathrm{KE}}]$. 137

Figure 2.21. Games defining correctness of hierarchical public-key encryption scheme HIBE. 141

Figure 2.22. Games defining CCA security of hierarchical public-key encryption scheme HIBE. 142

Figure 2.23. Key-updatable public-key encryption scheme $\mathsf{PKE} = \mathsf{PKE\text{-}CONS}[\mathsf{HIBE}]$. . 142

Figure 2.24. Adversary $\mathcal{A}_{\mathsf{HIBE}}$ used for proof of Theorem 12. 143

Figure 3.1. Canonical form of key exchange protocol. 158

Figure 3.2. Oracles specifying how the attacker can interact with OKE protocol KE. . . 161

Figure 3.3. Security game measuring the security of a messaging scheme Π when its keys are produced by OKE protocol KE. For compactness we define $\mathcal{K} = \textsc{Reg}, \textsc{Req}, \textsc{Fin}, \textsc{Fing}, \textsc{Corrupt}, \textsc{Reveal}$. 162

Figure 3.4. Real world in OKE security model. Some oracles are omitted because they are identical to previously specified oracles. 170

Figure 3.5. Ideal world in OKE security model. For compactness we let $\mathcal{O} = \textsc{IReg}, \textsc{IReq}, \textsc{IFin}, \textsc{IReveal}$. 171

Figure 3.6. Oracles of an ideal key exchange. 173

Figure 3.7. State exposure game with ideal key exchange. For compactness we let $\mathcal{O} = \text{IReg}, \text{IReq}, \text{IFin}, \text{IReveal}$. 178

Figure 3.8. Pseudocode formalizing the minimal V. 183

Figure 3.9. Pseudocode formalizing the maximal I. 184

Figure 3.10. Adversary for proof of Lemma 16 making at most one query to IFing. Oracles $\mathcal{O}\text{S}$ are described in the text. 186

Figure 3.11. Single user adversary for proof of Lemma 16. For compactness we let $\mathcal{O}\text{S} = \text{IRegS}, \text{IReqS}, \text{IFinS}, \text{IRevealS}, \text{IFingS}$. 190

Figure 3.12. Adversary $\mathcal{A}_{\mathsf{KE}}$ for proof of Theorem 15. For compactness we define $\mathcal{K}\text{S} = \text{RegS}, \text{ReqS}, \text{FinS}, \text{FingS}, \text{CorruptS}, \text{RevealS}$. 191

Figure 3.13. Additional oracles as simulated by $\mathcal{A}_{\mathsf{KE}}$. 192

Figure 3.14. Adversary $\mathcal{A}_{\Pi}$ for proof of Theorem 15. For compactness we define $\mathcal{K}\text{S} = \text{RegS}, \text{ReqS}, \text{FinS}, \text{FingS}, \text{CorruptS}, \text{RevealS}$. Additionally, we define $\mathcal{O} = \text{IReg}, \text{IReq}, \text{IFin}, \text{IReveal}$. 193

Figure 3.15. Security game for public key encryption. 196

Figure 3.16. Adversary $\mathcal{B}$ used for proof of Lemma 18. For compactness, “Require” statements are omitted. 197

Figure 3.17. Adversary $\mathcal{P}$ used for proof of Lemma 18. For compactness, “Require” statements are omitted. 198

Figure 3.18. Oracles shared by G_0 and G_1. 202

Figure 3.19. Oracles shared by G_0 and G_1. Boxed code is only included in G_1. 203

Figure 3.20. Reduction to security of PKE. 204

Figure 3.21. Additional oracles of reduction to security of PKE. 205

Figure 3.22. Oracles shared by G_2 and G_3. 208

Figure 3.23. Oracles shared by G_2 and G_3. Boxed code is only included in G_2. 209

Figure 3.24. Oracles of G_4. 214

Figure 3.25. Oracles of game H_0. 217

Figure 3.26. Oracles of game H_0. 218

Figure 3.27. Oracles of game H_0. 219

Figure 3.28. Oracles of game H_1. Highlighting indicates changes from H_0. 220

Figure 3.29. Oracles of game H_1. Highlighting indicates changes from H_0. 221

Figure 3.30. Oracles of game H_2. Highlighting indicates changes from H_1. Boxes indicated where code will change to become G_0. 222

Figure 3.31. Oracles of game H_2. Highlighting indicates changes from H_1. Boxes indicated where code will change to become G_0. 223

Figure 3.32. Reproduction of oracles of G_4. Highlighting indicates where code will change to become F_0. 224

Figure 3.33. Oracles of game G_4. Highlighting indicates where code will change to become F_0. 225

Figure 3.34. Oracles of game F_0. Boxes indicates changes from G_4. Highlighting indicates where code will change to become F_1. 226

Figure 3.35. Oracles of game F_0. Boxes indicates changes from G_4. Highlighting indicates where code will change to become F_1. 227

Figure 3.36. Oracles of game F_1. Highlighting indicates where code will change to become F_1. 228

Figure 3.37. Oracles of game F_2. 229

Figure 3.38. Oracles of game F_3. Highlighting indicates where KE was plugged into $G^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},1}$. Boxes indicate differences from F_2. 230

LIST OF TABLES

Table 2.1. Table summarizing some important variables in game AEAC. A “−” indicates a way in which the behavior of the adversary is being restricted. A “+” indicates a way in which the behavior of the adversary is being enabled. . . . 101

ACKNOWLEDGEMENTS

Thanks to my academic advisor, Mihir Bellare. His humanity, pursuit of precision, and knack for criticism are truly inspirational. His influence can be found in the many ways I have improved as a researcher and person these last five years.

Thanks to David Cash who introduced me to cryptography, convinced me to try research, and pushed my to pursue a PhD. Thanks to Tom Ristenpart and Stefano Tessaro for their mentorship during summer research visits and beyond.

Thanks to all the co-authors I have gotten to work with over the years. Particular thanks to Asha Camper Singh and Maya Nyayapati whose undergraduate research project was the seed out of which this dissertation bloomed and Igors Stepanovs for his co-authorship in my first paper without the direct guiding hand of a professor.

A PhD would be incomplete without many great colleagues and friends along the way. Attempting to list everyone would be folly, so I will arbitrarily restrict specific attention to two subsets. Thanks to my original officemates Karyn Benson, Dan Moeller, Igors Stepanovs, and Qiushi Wang for countless intriguing conversation. Thanks to Vivek Arte, Michael Borkowski, Juliana Curtis, Wei Dai, Hannah Davis, Nicholas Genise, Felix Günther, Matilda Lundin, Baiyu Li, Lucy Li, Julia Len, Ruth Ng, Mark Schultz, Jessica Sorrell, Igors Stepanovs, Jiahao Sun, Björn Tackmann, Michael Walter, and Qiushi Wang for companionship in cryptography. Thanks to all of the students, faculty, and staff that make UCSD CSE an amazing department.

A dissertation cannot exist without a committee to oversee it. Thanks to Massimo Franceschetti, Daniele Micciancio, Steven Savage, and Deian Stefan for taking the time to fulfill this role for mine.

Success in nearly any endeavor benefits from the support of loving friends and family. Thanks to my parents for always being there. Thanks to my brothers and cousins for providing much needed distraction and fun during breaks from school. Thanks to Emily Katz for all she has done.

Thanks to those I have inevitably missed.

Chapter 1, in full, is a reprint of the material as it appears in Advances in Cryptology - CRYPTO 2017. Bellare, Mihir; Camper Singh, Asha; Jaeger, Joseph; Nyayapati, Maya; Stepanovs, Igors, Springer Lecture Notes in Computer Science volume 10403, 2017. The dissertation author was the primary investigator and author of this paper. The dissertation author was supported in part by NSF grants CNS-1526801 and CNS-1228890

Chapter 2, in full, is a reprint of the material as it appears in Advances in Cryptology - CRYPTO 2018. Jaeger, Joseph; Stepanovs, Igors, Springer Lecture Notes in Computer Science volume 10991, 2018. The dissertation author was the primary investigator and author of this paper. The dissertation author was supported in part by NSF grants CNS-1717640 and CNS-1526801.

Chapter 3, in full, has been submitted for publication of the material as it may appear as "Key Exchange for Messaging Apps," Jaeger, Joseph. The dissertation author was the primary investigator and author of this paper. The dissertation author was supported in part by NSF grants CNS-1717640 and CNS-1526801.

VITA

2013	Bachelor of Arts, Rutgers University
2017	Master of Science, University of California San Diego
2019	Doctor of Philosophy, University of California San Diego

ABSTRACT OF THE DISSERTATION

Secure Messaging: From Systems to Theory

by

Joseph Jaeger

Doctor of Philosophy in Computer Science

University of California San Diego, 2019

Professor Mihir Bellare, Chair

The standard view of cryptography is that secure systems should be built by implementing known primitives whose theoretical security guarantees are well understood. In this work we take the opposite approach, taking inspiration from existing systems and providing the theoretical basis with which to understand their security goals. Our particular inspirations are modern secure messaging apps (e.g. Signal, WhatsApp) which have deployed new techniques with the goal of maintaining some security against attackers which sometimes gain temporary access to honest users' devices.

We propose that these security goals should be studied in a modular manner where distinct cryptographic components are studied in isolation. Towards this we separately provide

formal models for understanding the initial exchange of cryptographic secrets and their later use for the exchange of messages in this setting. We provide provable secure constructions of these separate components (often achieving better security than what is currently deployed by messaging apps) and a composition result which generically proves security when these isolated components are used together.

Introduction

End-to-end encrypted communication is becoming a usable reality for the masses in the form of secure messaging apps. End-to-end encryption provides a secure communication channel directly between two communicating users so that they can be assured that a service provider who is handling the transportation of messages between the two users can nonetheless not learn anything about what they are communicating.

However, chat sessions can be extremely long-lived and their secrets are stored on end user devices, so they are particularly vulnerable to having their cryptographic secrets exfiltrated to an attacker by malware or physical access to the device. If a traditional encryption protocol is used by the two communicating parties all security would now be lost. In a traditional protocol the two users typically permanently share a secret key K which is used to encrypt and decrypt messages. Given this permanent key an attacker would be able to read all past and future messages and impersonate either communicating party to the other at any point in time.

The Signal protocol [54] by Open Whisper Systems tries to mitigate this threat by continually updating the key used for encryption. Beyond its use in the Signal messaging app, this protocol has been adopted by a number of other secure messaging apps. This includes being used by default in WhatsApp and as part of secure messaging modes of Facebook Messenger, Google Allo, and Skype. WhatsApp alone has 1 billion daily active users [67].

The prominence of these systems raises the questions: what security are they aiming for, what security do they achieve, and could we do better? The answer to these questions does not seem clear. Indeed, in their Systemization of Knowledge paper on secure messaging, UDBFPGS [64] survey many of the systems that existed at the time and attempt to classify them

in terms of security, noting that security claims in different places include “forward-secrecy,” “backward-secrecy,” “self-healing” and “future secrecy,” and concluding that *“The terms are controversial and vague in the literature”* [64, Section 2.D].

In this work we take inspiration from these existing messaging systems and provide the theoretical basis with which to understand and answer these questions. We aim to be prescriptive, rather than descriptive. We aim to define the best possible security in the settings we consider and provide schemes which achieve it. While we are able to indicate where the existing deployed solutions fall short of this best possible security, we do not attempt to precisely capture what security is being achieved by them.

Security Goals.

At a high level, we can hope to achieve two sorts of security when a party’s secret state is compromised. First we desire that past messages remain secure when this occurs. This is the property often referred to as “forward-secrecy.” Second we desire that (as soon as possible) the security of future messages is not damaged. This property is referred to as “backward-secrecy,” “self-healing,” “future secrecy,” and “post-compromise security” among others. The precise definitions of these high level goals will depend on the primitive being studied.

Out definitions aim to capture privacy and integrity of communication. Informally, the former means that attackers should not be able to learn anything about what is being said in the conversation. The latter means that attackers cannot modify or insert messages in the conversation.

Modular Analysis.

In work done concurrently and independently to Chapter 1 of this dissertation, Cohn-Gordon, Cremers, Dowling, Garratt and Stebila [29] give a formal analysis of the Signal protocol. This analysis monolithically captures both the initial exchange of secrets between users to start a conversation and their later update during the conversation. The formal model in which they perform this analysis is extremely complex. At the same time it is somewhat unsatisfying because

they intentionally leave aside some security properties such as authenticity (which would require that an attacker cannot modify the conversation of two honest users) and do not model the actual exchange of messages. These shortcomings were likely motivated by a desire to avoid adding more complexity to their model.

Throughout this work we will continue to see that precise definitions capturing the threat of users' cryptographic secrets being exfiltrated repeatedly, but temporarily, are highly complex. To help tame this complexity, an important thesis of this work is that these security goals should be studied in a modular manner. Distinct cryptographic components are isolated and then studied independently. Then general composition results are provided which show that when these components are combined and used together, the overall system maintains the security of the underlying parts.

This approach provides a number of useful benefits. As noted already, it helps to tame complexity. The security of each component can first be understood in isolation without having to reference other components. In particular, this allows us to separately determine what is the best security we could hope for from each component. It also provides flexibility. If a messaging protocol is analyzed as a single monolithic entity then changing any subcomponent would require that the entire analysis be performed again. On the other hand, if the messaging protocol had been originally analyzed in a modular manner then only the particular cryptographic subcomponent which was modified needs to be studied again.

Cryptographic Components.

The primary division we make between cryptographic components of a messaging app is to separate the initial exchange of cryptographic secrets (i.e. the initial key exchange) from the later use of these secrets to exchange messages. For the former we must take into account the interacting behavior of all users simultaneously, while for the latter we can focus on a single messaging session between two users.

There are a variety of ways to divide the exchange of messages into multiple cryptographic

components. In this work we will look at two different options in this space. One is a technique-based approach in which we focus on what security can be achieved when using a particular technique for updating keys during communication. The other is a more goal-based approach.

The technique we focus on is ratcheting, which separates the update of secret keys from their use to encrypt communication. Borisov, Goldberg and Brewer (BGB) [22] introduced ratcheting in their highly influential OTR (Off the Record) communication system. (They do not call it ratcheting; this term originated later with Langley [47].) Ratcheting provides the core technique of how the Signal protocol [54] updates its keys over time. We provide the first definitions of ratcheting as an isolated cryptographic primitive. In particular we introduce the notions of ratcheted key exchange and ratcheted encryption. We exemplify the modularity of our definitions by showing that that secure ratcheted encryption can be generically obtained by combining secure ratcheted key exchange and a secure encryption scheme.

In our goal-based approach we consider (bidirectional) cryptographic channels which are the standard cryptographic models for bidirectional secure communication of messages. We first understand and then achieve the best possible security of cryptographic channels against an attacker that may arbitrarily and repeatedly learn the secret state of either communicating party.

Security Models.

In the security models throughout this work we typically think of the attacker as being the service provider itself. Thus we provide the attacker with complete control of communication between all of the honest users of the system. The attacker may arbitrarily read, block, modify, or re-order the communication between users. It is additionally given the ability to compromise the secret state of any user at any time. This leads to some inherent breaches of security because it can use this secret state to impersonate the user. We ask that security be maintained for all messages except those for which security is inherently breached.

There are numerous real world interpretations of who the attacker is that are captured by our models. The attacker may be the service provider itself (perhaps motivated by bad intent or

compelled into action by a government order). The attacker could be a malicious employee of the service provider acting without authorization. Alternatively, the attacker may be a hacker which has gained unauthorized access and control of the service provider's servers. Beyond this, it is well accepted practice in cryptography that we should assume attackers have as much control and access as possible. Trying to more precisely capture the limited access of an attacker can result in loss of security in practice if its capabilities have been underestimated.

Organization.

In Chapter 1 we propose the study of ratcheting as an isolated cryptographic primitive. Our intent is to formalize and understand the simplest form of ratcheting that captures the essence of the goal, which is one-sided ratcheting. In one-sided ratcheting we consider one-directional communication from a sender (assumed vulnerable to state compromise) to a receiver (assumed secure from state compromise). We provide a formal definition of security for one-sided ratcheted key exchange and one-sided ratcheted encryption. We provide a proven-secure construction of one-sided ratcheted key exchange (inspired by, but distinct from ratcheting techniques used in practice) and a proof that composing any secure one-sided ratcheted key exchange with a standard authenticated encryption scheme results in secure one-sided ratcheted encryption. The material presented in this chapter originally appeared in [17].

Chapter 2 takes the opposite approach. Rather than focusing on the particular technique of ratcheting we look at its goal, which is bidirectional messaging security against arbitrary and repeated compromise of communicating parties' states. We formalize the strongest possible security against this threat and provide a construction along with a proof that it achieves this security. Our construction makes use of new forms of asymmetric cryptographic primitives (key-updatable public-key encryption and key-updatable digital signature schemes) for which we provide definitions and constructions out of more standard cryptographic primitives. The material presented in this chapter originally appeared in [44].

Chapter 3 studies the key exchange used to share initial cryptographic secrets between

two users of a messaging app. Such a key exchange should be secure against state compromise in isolation but also when used to initiate a messaging protocol which provides mitigations against state compromise. In particular, if a messaging protocol is able to "heal" security after one of the party's cryptographic state has been revealed, then our key exchange should allow it to "heal" security even if the reveal happened during the execution of the key exchange. We provide an isolated definition for security of a key exchange protocol for a messaging app and a protocol which we prove meets this definition. Then we prove a composition result stating that a key exchange protocol which is secure according to our definition composes correctly with *any* underlying messaging protocol (including those presented in the earlier chapters of this dissertation and those of a number of related works [56, 35, 45, 2]). The material presented in this chapter is part of the manuscript "Key Exchange for Messaging App" by Joseph Jaeger, which has been submitted for publication.

Chapter 1

Ratcheted Encryption and Key Exchange

In this chapter, we aim to formalize the goals that ratcheting appears to be targeting. We give definitions for ratcheted encryption and ratcheted key-exchange. We then give protocols (based on ones in use but not identical to them) to provably achieve the goals.

This chapter aims to be selective rather than comprehensive. Our intent is to formalize and understand the simplest form of ratcheting that captures the essence of the goal, which is single, one-sided ratcheting. This (as we will see) is already complex enough.

Ratcheting.

The setting we consider is that sender Alice and receiver Bob hold keys $K_s = (k,\ldots)$ and $K_r = (k,\ldots)$, respectively, k representing a shared symmetric key and the ellipses indicating there may be more key information that may be party dependent. In practice, these keys are the result of a session-key exchange protocol that is authenticated either via the parties' certificates (TLS) or out-of-band (secure messaging); however, ratcheting is about how these keys are used and updated, not about how they are obtained, and so we will not be concerned with the distribution method, instead viewing the initial keys as created and distributed by a trusted process.

In TLS, all data is secured under the shared key k with an authenticated encryption scheme. Under ratcheting, the key is constantly changing. As per BGB [22] it works roughly like this:

$$B \rightarrow A\colon g^{b_1} \; ; \; A \rightarrow B\colon g^{a_1}, \mathcal{E}(k_1, M_1) \; ; \; B \rightarrow A\colon g^{b_2}, \mathcal{E}(k_2, M_2) \; ; \; \ldots \tag{1.1}$$

Here a_i and b_i are random exponents picked by A and B respectively; $k_1 = H(k, g^{b_1 a_1})$, $k_2 = H(k_1, g^{a_1 b_2})$, $\ldots$; H is a hash function; $\mathcal{E}$ is an encryption function taking key and message to return a ciphertext; and g is the generator of an underlying group. Each party deletes its exponents and keys once they are no longer needed for encryption or decryption.

Contributions.

This chapter aims to lift ratcheting from a technique to a cryptographic primitive, with a precise syntax and formally-defined security goals. Once this is done, we specify and prove secure some protocols that are closely related to the in-use ones.

If ratcheting is to be a primitive, a syntax is the first requirement. As employed, the ratcheting technique is used within a larger protocol, and one has to ask what it might mean in isolation. To allow a modular treatment, we decouple the creation of keys from their use, defining two primitives: ratcheted key exchange and ratcheted encryption. For each, we give a syntax. While ratcheting in apps is typically per message, our model is general and flexible, allowing the sender to ratchet the key at any time and encrypt as many messages as it likes under a given key before ratcheting again.

Next we give formal, game-based definitions of security for both ratcheted key exchange and ratcheted encryption. At the highest level, the requirement is that compromise (exposure in our model) revealing a party's current key and state should have only a local and temporary effect on security: a small hiccup, not compromising prior communications and after whose passage *both* privacy and integrity are somehow restored. This covers forward security (prior keys or communications remain secure) and backward security (future keys and communications remain secure). Amongst the issues in formalizing this is that following exposure there is some (necessary) time lag before security is regained, and that privacy and integrity are related. For ratcheted key exchange, un-exposed keys are required to be indistinguishable from random in the spirit of [15] —rather than merely, say, hard to recover— to allow them to be later securely used. For ratcheted encryption, the requirement is in the spirit of nonce-based authenticated

encryption [57], so that authenticity in particular is provided.

The definitions are chosen to allow a modular approach to constructions. We exemplify by showing how to build ratcheted encryption generically from ratcheted key-exchange and multi-user-secure nonce-based encryption [18]. This allows us to focus on ratcheted key exchange.

We give a protocol for ratcheted key exchange that is based on DH key exchanges. The core technique is the same as in [22] and the in-use protocols, but there are small but important differences, including MAC-based authentication of the key-update values and the way keys are derived. We prove that our protocol meets our definition of ratcheted key exchange under the SCDH (Strong Computational Diffie-Hellman) assumption [1] in the random oracle model (ROM) [14]. The proof is obtained in two steps. The first is a standard-model reduction to an assumption we call ODHE (Oracle Diffie-Hellman with Exposures). The second is a validation of ODHE under SCDH in the ROM.

Model and syntax.

Our syntax specifies a scheme RKE for ratcheted key exchange via three algorithms: initial key generation RKE.IKg, sender key generation RKE.SKg and receiver key generation RKE.RKg. See Fig. 1.3 for an illustration. The parties maintain output keys (representing the keys they are producing for an overlying application like ratcheted encryption) and session keys (local state for their internal use). At any time, the sender A can run RKE.SKg on its current keys to get update information upd that it sends to the receiver, as well as updated keys for itself. The receiver B correspondingly will run RKE.RKg on received update information and its current keys to get updated keys, transmitting nothing. RKE.IKg provides initial keys for the parties, what we called K_s and K_r above, that in particular contain an initial output key k (the same for both parties) and initial session keys. A ratcheted encryption scheme RE maintains the same three key-generation algorithms, now denoted RE.IKg, RE.SKg and RE.RKg, and adds an encryption algorithm RE.Enc for the sender —in the nonce-based vein [57], taking a key, nonce, message and header to deterministically return a ciphertext— and a corresponding decryption

algorithm RE.Dec for the receiver. The key for encryption and decryption is what ratcheted key exchange referred to as the output key.

Besides a natural correctness requirement, we have a robustness requirement: if the receiver receives an update that it rejects, it maintains its state and will still accept a subsequent correct update. This prevents a denial-of-service attack in which a single incorrect update sent to the receiver results in all future communications being rejected.

Security.

In the spirit of BR [15] we give the adversary complete control of communication. Our definition of security for ratcheted key exchange in Section 1.3.2 is via a game KIND. After (trusted) initial key-generation, the game gives the adversary oracles to invoke either sender or receiver key generation and also to expose sender keys (both output and session). Roughly the requirement is that un-exposed keys be indistinguishable from random. The delicate issue is that this is true only under some conditions. Thus, exposure in one session will compromise the next session. Also, a post-expose active attack on the receiver (in which the adversary supplies the update information) can result in continued violation of integrity. Our game makes the necessary restrictions to capture these and other situations. For ratcheted encryption, the game RAE we give in Section 1.4 captures ratcheted authenticated encryption with nonce-based security. The additional oracles for the adversary are encryption and decryption. The requirement is that, for un-exposed and properly restricted keys, the adversary cannot distinguish whether its encryption and decryption oracles are real, or return random ciphertexts and $\perp$ respectively.

Schemes.

Our ratcheted key exchange scheme in Section 1.3.3 is simple and efficient and uses the same basic DH technique as ratcheting in OTR [22] or WhatsApp, but analysis is quite involved. The sender's initial key includes g^b where b is part of the receiver's initial key, these quantities remaining static. Sender key generation algorithm RKE.SKg picks a random a and sends the update upd consisting of g^a *together with a mac under the prior session key* that is

crucial to security. The output and next session key are derived via a hash function applied to g^{ab}. Theorem 1 establishes that the scheme meets our stringent notion of security for ratcheted key exchange. The proof uses a game sequence that includes a hybrid argument to reduce the security of the ratcheted key exchange to our ODHE (Oracle Diffie-Hellman with Exposures) assumption. The latter is an extension of the ODH assumption of [1] and, like the latter, can be validated in the ROM under the SCDH assumption of [1] (which in turn is a variant of the Gap-DH assumption of [53]). We show this in Section 1.5. Ultimately, this yields a proof of security for our ratcheted key exchange protocol under the SCDH assumption in the ROM.

Our construction of a ratcheted encryption scheme in Section 1.4 is a generic combination of any ratcheted key exchange scheme (meeting our definition of security) and any nonce-based authenticated encryption scheme. Theorem 2 establishes that the scheme meets our notion of security for ratcheted encryption. The analysis is facilitated by assuming multi-user security for the base nonce-based encryption scheme as defined in [18], but a hybrid argument reduces this to the standard single-user security defined in [57]. Encryption schemes meeting this notion are readily available.

Setting and discussion.

There are many variants of ratcheting. What we treat is one-sided ratcheting. This means one party (Alice) is a sender and the other (Bob) a receiver, rather than both playing both roles. In our model, compromises (exposures) are allowed only on the sender, not on the receiver. In particular the receiver has a static secret key whose compromise will immediately violate privacy of our schemes, regardless of updates. From the application perspective, our model and schemes are suitable for settings where the sender (for example a smartphone) is vulnerable to compromise but the receiver (for example a server with hardware-protected storage) can keep keys safely. In two-sided ratcheting, both the sender and the receiver may be compromised. Another dimension is single (what we treat) versus double ratcheting. In the latter, keys are also locally ratcheted via a forward-secure pseudorandom generator [20]. Conceptually, we decided

to focus on the single, one-sided case to keep definitions (already quite complex) as simple as possible while capturing the essence of the goal and method. But we note that what Signal implements, and what is thus actually used, is double, two-sided ratcheting. Treating this does not seem like a simple extension of what we do and is left as future work.

Secure Internet communication protocols (both TLS and messaging) start with a session-key exchange that provides session keys, K_s for the sender and K_r for the receiver. These are our initial keys, the starting points for ratcheting. These keys are not to be confused with higher-level, long-lived signing or other keys that are certified either explicitly (TLS) or out-of-band (messaging) and used for authentication in the session-key exchange.

Messaging sessions tend to be longer lived than typical TLS sessions, with conversations that are on-going for months. This is part of why messaging security seeks, via ratcheting, fine-grained forward and backward security. Still, exactly what threat ratcheting prevents in practice needs careful consideration. If the threat is malware on a communicant's phone that can directly exfiltrate text of conversations, ratcheting will not help. Ratcheting will be of more help when users delete old messages, when the malware is exfiltrating keys rather than text, and when its presence on the phone is limited through software security.

Related work.

In concurrent and independent work, Cohn-Gordon, Cremers, Dowling, Garratt and Stebila (CCDGS) [29] give a formal analysis of the Signal protocol. The protocol they analyze includes ratcheting steps but stops at key distribution: unlike us, they do not consider, define or achieve ratcheted encryption. They treat Signal as a multi-stage session-key exchange protocol [38] in the tradition of authenticated session-key exchange [15, 12], with multiple parties and sessions. We instead consider ratcheted key exchange as a two-party protocol based on a trusted initial key distribution. This isolates ratcheted key exchange from the session key exchange used to produce the initial keys and allows a more modular treatment. They prove security (like us, in the ROM) under the Gap-DH [53] assumption while we prove it under the weaker

SCDH [1] assumption. Ultimately their work and ours have somewhat different goals. Theirs is to analyze the particular Signal protocol. Ours is to isolate the core ratcheting method (as one of the more novel elements of the protocol) and formalize primitives reflecting its goals in the simplest possible way.

Cohn-Gordon, Cremers and Garratt (CCG) [28] study and compare different kinds of post-compromise security in contexts including authenticated key exchange. They mention ratcheting as a technique for maintaining security in the face of compromise.

Key-insulated cryptography [32, 33, 34] also targets forward and backward security but in a model where there is a trusted helper and an assumed-secure channel from helper to user that is employed to update keys. Implementing the secure channel is problematic due to the exposures [7]. Ratcheting in contrast works in a model where all communication is under adversary control.

1.1 Preliminaries

Notation and conventions.

Let $\mathbb{N} = \{0, 1, 2, \ldots\}$ be the set of non-negative integers. Let ε denote the empty string. If $x \in \{0,1\}^*$ is a string then $|x|$ denotes its length, $x[i]$ denotes its i-th bit, and $x[i..j] = x[i] \ldots x[j]$ for $1 \leq i \leq j \leq |x|$. If mem is a table, we use $\mathsf{mem}[p]$ to denote the element of the table that is indexed by p. By $x \,\|\, y$ we denote a uniquely decodable concatenation of strings x and y (if lengths of x and y are fixed then $x \,\|\, y$ can be implemented using standard string concatenation). If X is a finite set, we let $x \leftarrow\$\, X$ denote picking an element of X uniformly at random and assigning it to x. We use a special symbol $\perp$ to denote an empty table position, and we also return it as an error code indicating an invalid input; we assume that adversaries never pass $\perp$ as input to their oracles.

Algorithms may be randomized unless otherwise indicated. Running time is worst case. If A is an algorithm, we let $y \leftarrow A(x_1, \ldots; r)$ denote running A with random coins r on inputs $x_1, \ldots$ and assigning the output to y. We let $y \leftarrow\$\, A(x_1, \ldots)$ be the result of picking r at random

Game $\mathrm{SUFCMA}_{\mathsf{F}}^{\mathcal{F}}$	Game $\mathrm{MAE}_{\mathsf{SE}}^{\mathcal{N}}$
$fk \leftarrow_{\$} \{0,1\}^{\mathsf{F.kl}}$ $\mathsf{win} \leftarrow \mathsf{false}$ $\mathcal{F}^{\textsc{Tag},\textsc{Verify}}$ Return win $\underline{\textsc{Tag}(m)}$ $\sigma \leftarrow \mathsf{F.Ev}(fk, m)$ $S \leftarrow S \cup \{(m, \sigma)\}$ Return σ $\underline{\textsc{Verify}(m, \sigma)}$ $\sigma' \leftarrow \mathsf{F.Ev}(fk, m)$ If $(\sigma = \sigma')$ and $((m, \sigma) \notin S)$ then $\mathsf{win} \leftarrow \mathsf{true}$ Return $(\sigma = \sigma')$	$b \leftarrow \{0,1\}$; $v \leftarrow 0$ $b' \leftarrow_{\$} \mathcal{N}^{\textsc{New},\textsc{Enc},\textsc{Dec}}$; Return $(b' = b)$ $\underline{\textsc{New}}$ $v \leftarrow v + 1$; $\mathsf{sk}[v] \leftarrow_{\$} \{0,1\}^{\mathsf{SE.kl}}$ $\underline{\textsc{Enc}(i, n, m, h)}$ If not $(1 \le i \le v)$ then return $\perp$ If $(i, n) \in U$ then return $\perp$ $c_1 \leftarrow \mathsf{SE.Enc}(\mathsf{sk}[i], n, m, h)$ $c_0 \leftarrow_{\$} \{0,1\}^{\mathsf{SE.cl}(\lvert m \rvert)}$ $U \leftarrow U \cup \{(i, n)\}$; $S \leftarrow S \cup \{(i, n, c_b, h)\}$ Return c_b $\underline{\textsc{Dec}(i, n, c, h)}$ If not $(1 \le i \le v)$ then return $\perp$ If $(i, n, c, h) \in S$ then return $\perp$ $m \leftarrow \mathsf{SE.Dec}(\mathsf{sk}[i], n, c, h)$ If $b = 1$ then return m else return $\perp$

Figure 1.1. Games defining strong unforgeability of function family F under chosen message attack, and multi-user authenticated encryption security of SE.

and letting $y \leftarrow A(x_1, \ldots; r)$. We let $[A(x_1, \ldots)]$ denote the set of all possible outputs of A when invoked with inputs $x_1, \ldots$. Adversaries are algorithms.

We use the code based game playing framework of [16]. (See Fig. 2.1 for an example.) We let Pr[G] denote the probability that game G returns true. In code, uninitialized integers are assumed to be initialized to 0, Booleans to false, strings to the empty string, sets to the empty set, and tables are initially empty.

Function families.

A family of functions F specifies a deterministic algorithm F.Ev. Associated to F is a key length $\mathsf{F.kl} \in \mathbb{N}$, an input set F.In, and an output length F.ol. Evaluation algorithm F.Ev takes $fk \in \{0,1\}^{\mathsf{F.kl}}$ and an input $x \in \mathsf{F.In}$ to return an output $y \in \{0,1\}^{\mathsf{F.ol}}$.

Strong unforgeability under chosen message attack.

Consider game SUFCMA of Fig. 1.1, associated to a function family F and an adversary $\mathcal{F}$. In order to win the game, adversary $\mathcal{F}$ has to produce a valid tag σ_{forge} for any message m_{forge}, satisfying the following requirement. The requirement is that $\mathcal{F}$ did not previously receive σ_{forge} as a result of calling its TAG oracle with m_{forge} as input. The advantage of $\mathcal{F}$ in breaking the SUFCMA security of F is defined as $\mathsf{Adv}^{\mathsf{sufcma}}_{\mathsf{F},\mathcal{F}} = \Pr[\mathrm{SUFCMA}^{\mathcal{F}}_{\mathsf{F}}]$. If no adversaries can achieve a high advantage in breaking the SUFCMA security of F while using only bounded resources, we refer to F as a MAC algorithm and we refer to its key fk as a MAC key.

Symmetric encryption schemes.

A symmetric encryption scheme SE specifies deterministic algorithms $\mathsf{SE.Enc}$ and $\mathsf{SE.Dec}$. Associated to SE is a key length $\mathsf{SE.kl} \in \mathbb{N}$, a nonce space $\mathsf{SE.NS}$, and a ciphertext length function $\mathsf{SE.cl}\colon \mathbb{N} \to \mathbb{N}$. Encryption algorithm $\mathsf{SE.Enc}$ takes $sk \in \{0,1\}^{\mathsf{SE.kl}}$, a nonce $n \in \mathsf{SE.NS}$, a message $m \in \{0,1\}^*$ and a header $h \in \{0,1\}^*$ to return a ciphertext $c \in \{0,1\}^{\mathsf{SE.cl}(|m|)}$. Decryption algorithm $\mathsf{SE.Dec}$ takes sk, n, c, h to return message $m \in \{0,1\}^* \cup \{\bot\}$, where $\bot$ denotes incorrect decryption. Decryption correctness requires that $\mathsf{SE.Dec}(sk, n, \mathsf{SE.Enc}(sk, n, m, h), h) = m$ for all $sk \in \{0,1\}^{\mathsf{SE.kl}}$, all $n \in \mathsf{SE.NS}$, all $m \in \{0,1\}^*$, and all $h \in \{0,1\}^*$. Nonce-based symmetric encryption was introduced in [58], whereas [57] also considers it in the setting with associated data. In this work we consider only *nonce-based* symmetric encryption schemes *with associated data*; we omit repeating these qualifiers throughout the text, instead referring simply to “symmetric encryption schemes”.

Multi-user authenticated encryption.

Consider game MAE of Fig. 1.1, associated to a symmetric encryption scheme SE and an adversary $\mathcal{N}$. It extends the definition of *authenticated encryption with associated data* for nonce-based schemes [57] to the multi-user setting, first formalized in [18]. The adversary is given access to oracles NEW, ENC and DEC. It can increase the number of users by calling oracle NEW, which generates a new (secret) user key. For any of the user keys, the adversary can

request encryptions of plaintext messages by calling oracle ENC and decryptions of ciphertexts by calling oracle DEC. In the real world (when $b = 1$), oracles ENC and DEC provide correct encryptions and decryptions. In the random world (when $b = 0$), oracle ENC returns uniformly random ciphertexts and oracle DEC returns the incorrect decryption symbol $\perp$. The goal of the adversary is to distinguish between these two cases. In order to avoid trivial attacks, $\mathcal{N}$ is not allowed to call DEC with ciphertexts that were returned by ENC. Likewise, we allow $\mathcal{N}$ to call ENC only once for every unique user-nonce pair (i,n). This can be strengthened to allow queries with repeated (i,n) and instead not allow queries with repeated (i,n,m,h), but the stronger requirement is satisfied by fewer schemes. The advantage of $\mathcal{N}$ in breaking the MAE security of SE is defined as $\mathsf{Adv}^{\mathsf{mae}}_{\mathsf{SE},\mathcal{N}} = 2\Pr[\mathrm{MAE}^{\mathcal{N}}_{\mathsf{SE}}] - 1$.

1.2 Oracle Diffie-Hellman with Exposures

The Oracle Diffie-Hellman (ODH) assumption [1] in a cyclic group requires that it is hard to distinguish between a random string and a hash function H applied to g^{xy}, even given g^x, g^y and an access to an oracle that returns $H(X^y)$ for arbitrary X (excluding $X = g^x$). We extend this assumption for multiple queries, based on a fixed g^y and arbitrarily many $g^{x[0]}, g^{x[1]}, \ldots$. For each index v we allow either to expose $x[v]$, or to get a challenge value; the challenge value is either a random string, or H applied to $g^{x[v]\cdot y}$. We also extend the hash function oracle to take a broader class of inputs.

Oracle Diffie-Hellman with Exposures assumption.

Let $\mathbb{G}$ be a cyclic group of order $p \in \mathbb{N}$, and let $\mathbb{G}^*$ denote the set of its generators. Let H be a function family such that $\mathsf{H.In} = \{0,1\}^*$. Consider game ODHE of Fig. 1.2 associated to $\mathbb{G}, \mathsf{H}$ and an adversary $\mathcal{O}$, where $\mathcal{O}$ is required to call oracle UP at least once prior to making any oracle queries to CH and EXP. The game starts by sampling a function key hk, a group generator g and a secret exponent y. The adversary is given hk, g, g^y and it has access to oracles UP, CH, EXP, HASH. Oracle UP generates a new challenge exponent $x[v]$ and returns $g^{x[v]}$,

$\underline{\text{Game ODHE}^{\mathcal{O}}_{\mathbb{G},\mathsf{H}}}$

$b \leftarrow\!\!\$ \{0,1\}$; $hk \leftarrow\!\!\$ \{0,1\}^{\mathsf{H.kl}}$; $g \leftarrow\!\!\$ \mathbb{G}^*$; $y \leftarrow\!\!\$ \mathbb{Z}_p$; $v \leftarrow -1$
$b' \leftarrow\!\!\$ \mathcal{O}^{\text{UP,CH,EXP,HASH}}(hk, g, g^y)$; Return $(b' = b)$

$\underline{\text{UP}}$

$\mathsf{op} \leftarrow \varepsilon$; $v \leftarrow v + 1$; $x[v] \leftarrow\!\!\$ \mathbb{Z}_p$; Return $g^{x[v]}$

$\underline{\text{CH}(s)}$

If $(\mathsf{op} =$ "exp") or $((v, s, g^{x[v]}) \in S_{\mathsf{hash}})$ then return $\perp$
$\mathsf{op} \leftarrow$ "ch" ; $S_{\mathsf{ch}} \leftarrow S_{\mathsf{ch}} \cup \{(v, s, g^{x[v]})\}$; $e \leftarrow g^{x[v] \cdot y}$
If $\mathsf{mem}[v, s, e] = \perp$ then $\mathsf{mem}[v, s, e] \leftarrow\!\!\$ \{0,1\}^{\mathsf{H.ol}}$
$r_1 \leftarrow \mathsf{H.Ev}(hk, v \,\|\, s \,\|\, e)$; $r_0 \leftarrow \mathsf{mem}[v, s, e]$; Return r_b

$\underline{\text{EXP}}$

If $\mathsf{op} =$ "ch" then return $\perp$
$\mathsf{op} \leftarrow$ "exp" ; Return $x[v]$

$\underline{\text{HASH}(i, s, X)}$

If $(i, s, X) \in S_{\mathsf{ch}}$ then return $\perp$
If $i = v$ then $S_{\mathsf{hash}} \leftarrow S_{\mathsf{hash}} \cup \{(i, s, X)\}$
Return $\mathsf{H.Ev}(hk, i \,\|\, s \,\|\, X^y)$

Figure 1.2. Game defining Oracle Diffie-Hellman with Exposures assumption for $\mathbb{G}, \mathsf{H}$.

---

where v is an integer counter that denotes the number of the current challenge exponent (indexed from 0) and is incremented by 1 at the start of every call to oracle UP. Oracle HASH takes an arbitrary integer i, an arbitrary string s and a group element X to return $\mathsf{H.Ev}(hk, i \,\|\, s \,\|\, X^y)$. For each counter value v, the adversary can choose to either call oracle EXP to get the value of $x[v]$ or call oracle CH with input s to get a challenge value that is generated as follows. In the real world (when $b = 1$) oracle CH returns $\mathsf{H.Ev}(hk, v \,\|\, s \,\|\, g^{x[v] \cdot y})$ and in the random world (when $b = 0$) it returns a uniformly random element from $\{0,1\}^{\mathsf{H.ol}}$. The goal of the adversary is to distinguish between these two cases. Oracle CH can be called multiple times per challenge exponent, and it returns consistent outputs regardless of the challenge bit's value. The advantage of $\mathcal{O}$ in breaking the ODHE security of $\mathbb{G}, \mathsf{H}$ is defined as $\mathsf{Adv}^{\mathsf{odhe}}_{\mathbb{G},\mathsf{H},\mathcal{O}} = 2\Pr[\text{ODHE}^{\mathcal{O}}_{\mathbb{G},\mathsf{H}}] - 1$.

In order to avoid trivial attacks, $\mathcal{O}$ is not allowed to query oracle HASH on input (i, s, X) if $X = g^{x[i]}$ and if oracle CH was already called with input s when the counter value was $v = i$. Note that adversary is allowed to win the game if it happens to guess a future challenge exponent

x and query it to oracle HASH ahead of time; the corresponding triple (i,s,X) will not be added to the set of inputs S_{hash} that are not allowed to be made to oracle CH. Finally, recall that the string concatenation operator $\|$ is defined to produce uniquely decodable strings, which helps to avoid trivial string padding attacks.

Plausibility of the ODHE assumption.

We do not know of any group $\mathbb{G}$ and function family H that can be shown to achieve ODHE in the standard model. The original ODH assumption of [1] was justified by a reduction in the random oracle model to the Strong Computational Diffie-Hellman (SCDH) assumption. The latter was defined in [1] and is a weaker version of the Gap Diffie-Hellman assumption from [53]. In Section 1.5 we give a definition for the SCDH assumption and prove that it also implies the ODHE assumption in the random oracle model.

We provide this result as a corollary of two lemmas. The lemmas use the Strong Computational Diffie-Hellman with Exposures (SCDHE) assumption as an intermediate step, where SCDHE is a novel assumption that extends SCDH to allow multiple challenge queries, and to allow exposures. To formalize our result, we define the Oracle Diffie-Hellman with Exposures in ROM (ODHER) assumption that is equivalent to the ODHE assumption in the random oracle model.

The first lemma establishes that SCDHE implies ODHE in the random oracle model, by a reduction from ODHER to SCDHE. The proof of this lemma emulates the ODH to SCDH reduction of [1]. In their reduction, the SCDH adversary simulates the random oracle and the hash oracle for the ODH adversary; it uses its own decisional-DH oracle to check whether the ODH adversary feeds g^{xy} for the challenge values of x and y, and to maintain consistency between simulated oracle outputs. This consistency maintenance is the main source of complexity in our reduction because —in addition to the oracles mentioned above— we must also ensure that the simulated challenge oracle is consistent.

The second lemma is a standard model reduction from SCDHE to SCDH. This reduction

is a standard “guess the index” reduction in which our SCDH adversary guesses which query the SCDHE adversary will attack. The SCDH adversary replaces the answer to this query with the challenge values it was given and replaces all other oracle queries with challenges that it has generated itself. As usual, this results in a multiplicative loss of security, so the final theorem (combining both lemmas) has a bound of the form $\mathsf{Adv}^{\mathsf{odher}}_{\mathbb{G},\mathsf{H},\mathcal{O}} \leq q_{\textsc{Up}} \cdot \mathsf{Adv}^{\mathsf{scdh}}_{\mathbb{G},\mathcal{S}} + q_{\textsc{Hash}}/p$, where $\mathcal{S}$ is the SCDH adversary and $q_{\textsc{Up}}$ (resp. $q_{\textsc{Hash}}$) is the number of $\textsc{Up}$ (resp. $\textsc{Hash}$) queries made by ODHER adversary $\mathcal{O}$.

Because of the multiplicative loss of security caused by the second lemma we examine the possibility of using Diffie-Hellman self-reducibility techniques to obtain a tighter bound on the reduction from SCDHE to SCDH. The possibility of exposures in SCDHE makes this much more difficult than one might immediately realize. We present a reduction that succeeds despite these difficulties by using significantly more complicated methods than in our first example of this reduction. Specifically we build an SCDH adversary that makes guesses about the future behavior of the SCDHE adversary it was given and “rewinds” this adversary whenever its guess was incorrect. Thus we ultimately obtain the tighter bound of $\mathsf{Adv}^{\mathsf{odher}}_{\mathbb{G},\mathsf{H},\mathcal{O}} \leq \mathsf{Adv}^{\mathsf{scdh}}_{\mathbb{G},\mathcal{S}_u} + q_{\textsc{Up}} \cdot 2^{-u} + q_{\textsc{Hash}}/p$. Here $\mathcal{S}_u$ is the SCDH adversary that is defined for any parameter $u \in \mathbb{N}$ that bounds its worst case running time.

1.3 Ratcheted key exchange

Ratcheted key exchange allows users to agree on shared secret keys while providing very strong security guarantees. In this work we consider a setting that encompasses two parties, and we assume that only one of them sends key agreement messages. We call this party a sender, and the other party a receiver. This model enables us to make the first steps towards capturing the schemes that are used in the real world messaging applications. Future work could extend our model to allow both parties to send key agreement messages, and to consider the group chat setting where multiple users engage in shared conversations.

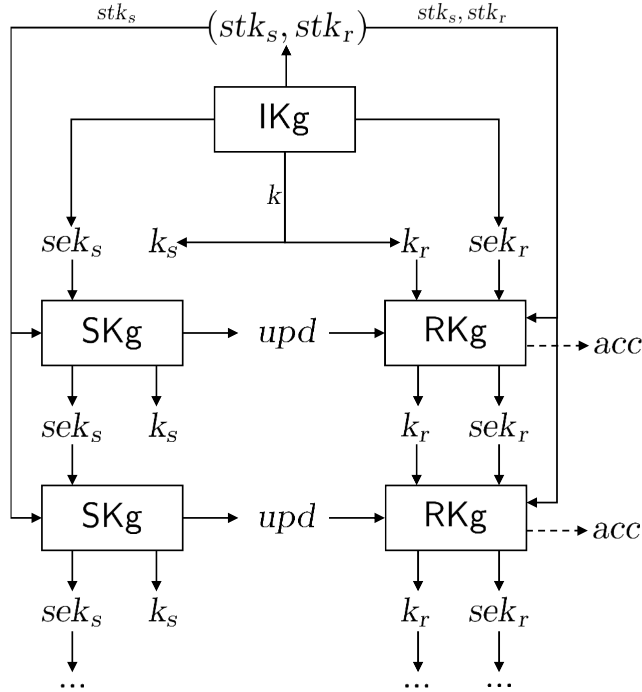

Figure 1.3. The interaction between ratcheted key exchange algorithms.

1.3.1 Definition of ratcheted key exchange

Consider Fig. 1.3 for an overview of algorithms that constitute a racheted key exchange scheme RKE, and the interaction between them. The algorithms are RKE.IKg, RKE.SKg and RKE.RKg. We will first provide an informal description of their functionality, and then formalize their syntax and correctness requirements.

Initial key generation algorithm RKE.IKg generates and distributes the following keys: $k, stk_s, stk_r, sek_s, sek_r$. Output key k is the initial shared secret key that can be used by both parties for any purpose such as running a symmetric encryption scheme. Static keys stk_s and stk_r are long-term keys that will not get updated over time. It is assumed that stk_s is known to all parties, whereas stk_r contains potentially secret information and will be known only by the receiver. Session keys sek_s and sek_r contain secret information that is required for future key exchanges, such as MAC keys (to ensure the authenticity of key exchange) and temporary secrets (that could be used for the generation of the next output keys). As a result of running RKE.IKg,

the sender gets stk_s, sek_s, k_s and the receiver gets stk_s, stk_r, sek_r, k_r, where $k_s = k_r = k$. We use "s" and "r" as subscripts for output keys and session keys, to indicate that the particular key is owned by the sender or by the receiver, respectively. Note that normally both parties will have the same output key (i.e. $k_s = k_r$), but this might not be true if an attacker succeeds to tamper with the protocol.

Next we define the sender's and receiver's key generation algorithms $\mathsf{RKE.SKg}$ and $\mathsf{RKE.RKg}$. These algorithms model the key ratcheting process that generates new session keys and output keys while deleting the corresponding old keys.

Sender's key generation algorithm $\mathsf{RKE.SKg}$ is run whenever the sender wants to produce a new shared secret key. It takes the sender's static key stk_s and the sender's session key sek_s. It returns an updated sender's session key sek_s, a new output key k_s, and update information upd. The update information is used by the receiver to generate the same output key.

Receiver's key generation algorithm $\mathsf{RKE.RKg}$ takes sender's static key stk_s, receiver's static key stk_r, receiver's session key sek_r, update information upd (received from the sender) and the current shared output key k_r. It returns receiver's session key sek_r, output key k_r, and a Boolean flag acc indicating whether the new keys were generated succesfully. Setting $acc = \mathsf{false}$ will generally mean that the received update information was rejected; our correctness definition will require that in such case the receiver's output key k_r and the receiver's session key sek_r should remain unchanged. This requirement is the reason why $\mathsf{RKE.RKg}$ takes the old value of k_r as one of its inputs.

Ratcheted key exchange schemes.

A ratcheted key exchange scheme RKE specifies algorithms $\mathsf{RKE.IKg}$, $\mathsf{RKE.SKg}$ and $\mathsf{RKE.RKg}$. Associated to RKE is an output key length $\mathsf{RKE.kl} \in \mathbb{N}$ and sender's key generation randomness space $\mathsf{RKE.RS}$. Initial key generation algorithm $\mathsf{RKE.IKg}$ returns $k, sek_s, (stk_s, stk_r, sek_r)$, where $k \in \{0,1\}^{\mathsf{RKE.kl}}$ is an output key, sek_s is a sender's session key, and stk_s, stk_r, sek_r are sender's static key, receiver's static key and receiver's session key, respec-

Game RKE-COR$_{\mathsf{R}}^{\mathcal{C}}$	Game RE-COR$_{\mathsf{R}}^{\mathcal{C}}$
$\mathsf{bad} \leftarrow \mathsf{false}$ $(k, sek_s, (stk_s, stk_r, sek_r)) \leftarrow \mathsf{R.IKg}$ $k_s \leftarrow k\,;\ k_r \leftarrow k$ $\mathcal{C}^{\textsc{Up},\textsc{RatRec}}\,;$ Return $(\mathsf{bad} = \mathsf{false})$	$\mathsf{bad} \leftarrow \mathsf{false}$ $(k, sek_s, (stk_s, stk_r, sek_r)) \leftarrow \mathsf{R.IKg}$ $k_s \leftarrow k\,;\ k_r \leftarrow k$ $\mathcal{C}^{\textsc{Up},\textsc{RatRec},\textsc{Enc}}\,;$ Return $(\mathsf{bad} = \mathsf{false})$

<u>$\textsc{Up}$</u>
$r \leftarrow\$ \mathsf{R.RS}\,;\ (sek_s, k_s, upd) \leftarrow \mathsf{R.SKg}(stk_s, sek_s; r)$
$(sek_r, k_r, acc) \leftarrow\$ \mathsf{R.RKg}(stk_s, stk_r, sek_r, upd, k_r)$
If not $((acc = \mathsf{true})$ and $(k_s = k_r))$ then $\mathsf{bad} \leftarrow \mathsf{true}$

<u>$\textsc{RatRec}(upd)$</u>
$(sek_r', k_r', acc) \leftarrow\$ \mathsf{R.RKg}(stk_s, stk_r, sek_r, upd, k_r)$
If $(acc = \mathsf{false})$ and not $((k_r' = k_r)$ and $(sek_r' = sek_r))$ then $\mathsf{bad} \leftarrow \mathsf{true}$

<u>$\textsc{Enc}(n, m, h)$</u>
$c \leftarrow \mathsf{R.Enc}(k_s, n, m, h)\,;\ m' \leftarrow \mathsf{R.Dec}(k_r, n, c, h)\,;$ If $(m' \neq m)$ then $\mathsf{bad} \leftarrow \mathsf{true}$

Figure 1.4. Game RKE-COR defining correctness of ratcheted key exchange scheme R, and game RE-COR defining correctness of ratcheted encryption scheme R. Oracles $\textsc{Up}$ and $\textsc{RatRec}$ are used in both games, whereas oracle $\textsc{Enc}$ is only used in game RE-COR.

---

tively. The sender's and receiver's output keys are initialized to $k_s = k_r = k$. Sender's key generation algorithm RKE.SKg takes stk_s, sek_s and randomness $r \in \mathsf{RKE.RS}$ to return a new sender's session key sek_s, a new sender's output key $k_s \in \{0,1\}^{\mathsf{RKE.kl}}$, and update information upd. Receiver's key generation algorithm RKE.RKg takes stk_s, stk_r, sek_r, upd and receiver's output key $k_r \in \{0,1\}^{\mathsf{RKE.kl}}$ to return a new receiver's session key sek_r, a new receiver's output key $k_r \in \{0,1\}^{\mathsf{RKE.kl}}$, and a flag $acc \in \{\mathsf{true}, \mathsf{false}\}$.

Correctness of ratcheted key exchange.

Consider game RKE-COR of Fig. 1.4 associated to a ratcheted key exchange scheme R and an adversary $\mathcal{C}$, where $\mathcal{C}$ is provided with an access to oracles $\textsc{Up}$ and $\textsc{RatRec}$.

Oracle $\textsc{Up}$ runs algorithm R.SKg to generate a new sender's output key k_s along with the corresponding update information upd; it then runs R.RKg with upd as input to generate a new receiver's output key k_r. It is required that $acc = \mathsf{true}$ and $k_s = k_r$ at the end of every $\textsc{Up}$ call. This means that if the receiver uses update information received from the sender (in the correct order), it is guaranteed to successfully generate the same output keys as the sender.

Oracle $\textsc{RatRec}$ takes update information upd of adversary's choice and attempts to run $\mathsf{R.RKg}$ with upd (and current receiver's keys) as input. The correctness requires that if the receiver's key update fails (meaning $acc = \mathsf{false}$) then the receiver's keys k_r, sek_r remain unchanged. This means that if receiver's attempt to generate new keys is not successful (e.g. if the update information is corrupted in transition), then the receiver's key generation algorithm should not corrupt the receiver's current keys. This is a usability property that requires that it is possible to recover from failures, meaning that the receiver can later re-run its key generation algorithm with the correct update information to successfully produce its next pair of (session and output) keys.

We consider an unbounded adversary and allow it to call its oracles in any order. The advantage of $\mathcal{C}$ breaking the correctness of R is defined as $\mathsf{Adv}^{\mathsf{rkecor}}_{\mathsf{R},\mathcal{C}} = 1 - \Pr[\text{RKE-COR}^{\mathcal{C}}_{\mathsf{R}}]$. Correctness property requires that $\mathsf{Adv}^{\mathsf{rkecor}}_{\mathsf{R},\mathcal{C}} = 0$ for all unbounded adversaries $\mathcal{C}$. Note that our definition of the correctness game with an unbounded adversary is equivalent to a more common correctness definition that would instead explicitly quantify over all randomness choices of all algorithms. We stress that our correctness definition does *not* require any security properties. In particular, it does not require that the update information is authenticated because oracle $\textsc{RatRec}$ considers only the case when $\mathsf{R.RKg}$ sets $acc = \mathsf{false}$.

Our definition requires *perfect* correctness. However, it can be relaxed by requiring that adversary $\mathcal{C}$ can only make a bounded number of calls to its oracles, and further requiring that its advantage of winning the game is negligible.

1.3.2 Security of ratcheted key exchange

Ratcheted key exchange attempts to provide strong security guarantees even in the presence of an attacker that can steal the secrets stored by the sender. Specifically, we consider an active attacker that is able to intercept and modify any update information sent from the sender to the receiver. The goal is that the attacker cannot distinguish the produced output keys from random strings, and cannot make the two parties agree on output keys that do not match.

Furthermore, we desire certain stronger security properties to hold even if the attacker manages to steal secrets stored by the sender, which we refer to as forward security and backward security. Forward security requires that such an attacker cannot distinguish prior keys from random. Backward security requires that the knowledge of sender's secrets at the current time period can not be used to distinguish keys generated (at some near point) in the future from random strings. Recall that our model is intentionally one-sided; exposure of receiver's secrets is not allowed. In particular, compromise of all of the receiver's secrets will permanently compromise security.

It is clear that if an attacker steals the secret information of the sender, then it can create its own update information resulting in the receiver agreeing on a "secret" key that is known by the attacker. It can be difficult to say what restrictions should be placed on the keys that the attacker makes the receiver agree to. Is it a further breach of security if the attacker then later causes the sender and the receiver to agree on the same secret key? What should happen if the attacker later forwards update information that was generated by the sender to the receiver?

In our security model we choose to insist on two straightforward policies in this scenario. The first is that whenever update information not generated by the sender is accepted by the receiver, even full knowledge of the key that the receiver has generated should not leak any information about other correctly generated keys. The second is that at any fixed point in time, if update information generated by the sender is accepted by the receiver then the receiver should agree with the sender on what the corresponding output key is, and the adversary should not be able to distinguish the shared output key from random.

Key indistinguishability of ratcheted key exchange schemes.

Consider game KIND on the left side of Fig. 1.5 associated to a ratcheted key exchange scheme RKE and an adversary $\mathcal{D}$. The advantage of $\mathcal{D}$ in breaking the KIND security of RKE is defined as $\mathsf{Adv}^{\mathsf{kind}}_{\mathsf{RKE},\mathcal{D}} = 2\Pr[\mathrm{KIND}^{\mathcal{D}}_{\mathsf{RKE}}] - 1$.

The adversary is given the sender's static key stk_s as well as access to oracles $\textsc{RatSend}$, $\textsc{RatRec}$, $\textsc{Exp}$, $\textsc{ChSend}$, and $\textsc{ChRec}$. It can call oracle $\textsc{RatSend}$ to receive update infor-

mation upd from the sender, and it can call oracle RATREC to pass arbitrary update information to the receiver. Oracle EXP returns the current secrets sek_s, k_s possessed by the sender as well as the random seed *r* that was used to create the most recent upd in RATSEND. Note that according to our notation convention from Section 1.1, integer variable *r* is assumed to be initialized to 0 at the beginning of the security game; this value will be returned if adversary calls EXP prior to RATSEND.

The challenge oracles CHSEND and CHREC provide the adversary with keys k_s and k_r in the real world (when $b = 1$), or with uniformly random bit strings in the random world (when $b = 0$). The goal of the adversary is to distinguish between these two worlds. To disallow trivial attacks the game makes use of tables op and auth (initialized as empty) as well as a boolean flag restricted (initialized as false). Specifically, op keeps track of the oracle calls made by the adversary and is used to ensure that it can not trivially win the game by calling oracle EXP to get secrets that were used for one of the challenge queries. Table auth keeps track of the update information upd generated by RATSEND so that we can set the flag restricted whenever the adversary has taken advantage of an EXP query to send maliciouly generated upd to RATREC. In this case we do not expect the receiver's key k_r to look random or match the sender's key k_s so CHREC is "restricted" and will return k_r in both the real and random worlds.

Authenticity of key exchange.

Our security definition implicitly requires the authenticity of key exchange. Specifically, assume that an adversary can violate the authenticity in a non-trivial way, meaning without using EXP oracle to acquire the relevant secrets. This means that the adversary can construct malicious update information upd^* that is accepted by the receiver, while not setting the restricted flag to true. By making the receiver accept upd^*, the adversary achieves the situation when the sender and the receiver produce different output keys $k_s \neq k_r$. Now adversary can call oracles CHSEND and CHREC to get both keys and compare them to win the game. In the real world ($b = 1$) the returned keys will be different, whereas in the random world ($b = 0$) they will be the same. We

formalize this attack in Section 1.6.

Allowing recovery from failures.

Consider a situation when an attacker steals all sender's secrets, and hence has an ability to impersonate the sender. It can drop all further packets sent by the sender and instead use the exposed secrets to agree on its own shared secret keys with the receiver. In the security game this corresponds to the case when the adversary calls EXP and then starts calling oracle RATREC with maliciously generated update information upd. This sets the restricted flag to true, making the CHREC oracle always return the real receiver's key k_r regardless of the value of game's challenge bit b. The design decision at this point is – do we want to allow the game to recover from this state, meaning should the restricted flag be ever set back to false?

Our decision on this matter was determined by the two "policies" discussed above. As long as the adversary keeps sending maliciously generated update information upd, the restricted flag will remain true. In this case, the real receiver's key k_r returned from CHREC should be of no help in distinguishing the real sender's key k_s from random, as desired from the first policy. To match the second policy, the next time adversary forwards the upd generated by the sender (i.e. $upd = \mathsf{auth}[i_r]$) to RATREC, if upd is accepted by the receiver then the restricted flag is set back to false. This makes the output of CHREC again depend on the challenge bit, thus requiring k_r to be equal to k_s and indistinguishable from random.

Alternative treatment of restricted **flag.**

Our security definition of KIND can be strengthened by making it never reset the restricted flag back to false. Instead, the game could require that if the adversary exposes sender's secrets and uses them to agree on its own shared output key with the receiver, then all the communication between the sender and the receiver should be disrupted. Meaning that any future attempt to simply forward sender's update information upd to the receiver should result in RATREC rejecting it. Otherwise adversary would be defined to win the game. This can be formalized in a number of ways. Our construction of ratcheted key exchange from Section 1.3.3

Game $\text{KIND}_{\mathsf{RKE}}^{\mathcal{D}}$

$b \leftarrow\$ \{0,1\}$; $i_s \leftarrow 0$; $i_r \leftarrow 0$
$(k, sek_s, (stk_s, stk_r, sek_r)) \leftarrow\$ \mathsf{RKE.IKg}$
$k_s \leftarrow k$; $k_r \leftarrow k$
$b' \leftarrow\$ \mathcal{D}^{\text{RatSend},\text{RatRec},\text{Exp},\text{ChSend},\text{ChRec}}(stk_s)$
Return $(b' = b)$

<u>RatSend</u>

$r \leftarrow\$ \mathsf{RKE.RS}$
$(sek_s, k_s, upd) \leftarrow \mathsf{RKE.SKg}(stk_s, sek_s; r)$
$\mathsf{auth}[i_s] \leftarrow upd$; $i_s \leftarrow i_s + 1$
Return upd

<u>RatRec(upd)</u>

$z \leftarrow\$ \mathsf{RKE.RKg}(stk_s, stk_r, sek_r, upd, k_r)$
$(sek_r, k_r, acc) \leftarrow z$
If not acc then return false
If $\mathsf{op}[i_r] =$ "exp" then restricted $\leftarrow$ true
If $upd = \mathsf{auth}[i_r]$ then restricted $\leftarrow$ false
$i_r \leftarrow i_r + 1$; Return true

<u>Exp</u>

If $\mathsf{op}[i_s] =$ "ch" then return $\perp$
$\mathsf{op}[i_s] \leftarrow$ "exp" ; Return (r, sek_s, k_s)

<u>ChSend</u>

If $\mathsf{op}[i_s] =$ "exp" then return $\perp$
$\mathsf{op}[i_s] \leftarrow$ "ch"
If $\mathsf{rkey}[i_s] = \perp$ then $\mathsf{rkey}[i_s] \leftarrow\$ \{0,1\}^{\mathsf{RKE.kl}}$
If $b = 1$ then return k_s else return $\mathsf{rkey}[i_s]$

<u>ChRec</u>

If restricted then return k_r
If $\mathsf{op}[i_r] =$ "exp" then return $\perp$
$\mathsf{op}[i_r] \leftarrow$ "ch"
If $\mathsf{rkey}[i_r] = \perp$ then $\mathsf{rkey}[i_r] \leftarrow\$ \{0,1\}^{\mathsf{RKE.kl}}$
If $b = 1$ then return k_r else return $\mathsf{rkey}[i_r]$

Game $\text{RAE}_{\mathsf{RE}}^{\mathcal{A}}$

$b \leftarrow\$ \{0,1\}$; $i_s \leftarrow 0$; $i_r \leftarrow 0$
$(k, sek_s, (stk_s, stk_r, sek_r)) \leftarrow\$ \mathsf{RE.IKg}$
$k_s \leftarrow k$; $k_r \leftarrow k$
$b' \leftarrow\$ \mathcal{A}^{\text{RatSend},\text{RatRec},\text{Exp},\text{Enc},\text{Dec}}(stk_s)$
Return $(b' = b)$

<u>RatSend</u>

$r \leftarrow\$ \mathsf{RE.RS}$
$(sek_s, k_s, upd) \leftarrow \mathsf{RE.SKg}(stk_s, sek_s; r)$
$\mathsf{auth}[i_s] \leftarrow upd$; $i_s \leftarrow i_s + 1$
Return upd

<u>RatRec(upd)</u>

$z \leftarrow\$ \mathsf{RE.RKg}(stk_s, stk_r, sek_r, upd, k_r)$
$(sek_r, k_r, acc) \leftarrow z$
If not acc then return false
If $\mathsf{op}[i_r] =$ "exp" then restricted $\leftarrow$ true
If $upd = \mathsf{auth}[i_r]$ then restricted $\leftarrow$ false
$i_r \leftarrow i_r + 1$; Return true

<u>Exp</u>

If $\mathsf{op}[i_s] =$ "ch" then return $\perp$
$\mathsf{op}[i_s] \leftarrow$ "exp" ; Return (r, sek_s, k_s)

<u>Enc(n, m, h)</u>

If $\mathsf{op}[i_s] =$ "exp" then return $\perp$
$\mathsf{op}[i_s] \leftarrow$ "ch"
If $(i_s, n) \in U$ then return $\perp$
$c_1 \leftarrow \mathsf{RE.Enc}(k_s, n, m, h)$
$c_0 \leftarrow\$ \{0,1\}^{\mathsf{RE.cl}(|m|)}$; $U \leftarrow U \cup \{(i_s, n)\}$
$S \leftarrow S \cup \{(i_s, n, c_b, h)\}$
Return c_b

<u>Dec(n, c, h)</u>

If restricted then
  Return $\mathsf{RE.Dec}(k_r, n, c, h)$
If $\mathsf{op}[i_r] =$ "exp" then return $\perp$
$\mathsf{op}[i_r] \leftarrow$ "ch"
If $(i_r, n, c, h) \in S$ then return $\perp$
$m \leftarrow \mathsf{RE.Dec}(k_r, n, c, h)$
If $b = 1$ then return m else return $\perp$

Figure 1.5. Games defining key indistinguishability of ratcheted key exchange scheme RKE, and authenticated encryption security of ratcheted encryption scheme RE.

should be secure for a stronger definition like that, but would likely require stronger assumptions to prove.

1.3.3 Construction of a ratcheted key exchange scheme

In this section we construct a ratcheted key exchange scheme, and discuss some design considerations by presenting a number of attacks that our scheme manages to evade. In Section 1.3.4 we will deduce a bound on the success of any adversary attacking the KIND security of our scheme. The idea of our construction is as follows. We let the sender and the receiver perform the Diffie-Hellman key exchange. The receiver's static key contains a secret DH exponent $stk_r = y$ and the sender's static key contains the corresponding public value $stk_s = g^y$ (working in some cyclic group with generator g). In order to generate a new shared secret key, the sender picks its own secret exponent x and computes the output key (roughly) as $k_s = H(stk_s^x) = H(g^{xy})$, where H is some hash function. The sender then sends update information containing g^x to the receiver, enabling the latter to compute the same output key. In order to ensure the security of the key exchange, both parties use a shared MAC key, meaning the update information also includes a tag of g^x.

Note that the used MAC key should be regularly renewed in order to ensure that the scheme provides backward security against exposures. As a result, the output of applying the hash function on g^{xy} is also used to derive a new MAC key. The initial key generation provides both parties with a shared MAC key and a shared secret key that are sampled uniformly at random. The formal definition of our key exchange scheme is as follows.

Ratcheted key exchange scheme $\mathsf{RATCHET\text{-}KE}$.

Let $\mathbb{G}$ be a cyclic group of order $p \in \mathbb{N}$, and let $\mathbb{G}^*$ denote the set of its generators. Let F be a function family such that $\mathsf{F.In} = \mathbb{G}$. Let H be a function family such that $\mathsf{H.In} = \{0,1\}^*$ and $\mathsf{H.ol} > \mathsf{F.kl}$. We build a ratcheted key exchange scheme $\mathsf{RKE} = \mathsf{RATCHET\text{-}KE}[\mathbb{G}, \mathsf{F}, \mathsf{H}]$ as defined in Fig. 1.6, with $\mathsf{RKE.kl} = \mathsf{H.ol} - \mathsf{F.kl}$ and $\mathsf{RKE.RS} = \mathbb{Z}_p$.

Algorithm $\mathsf{RKE.IKg}$	Algorithm $\mathsf{RKE.SKg}((hk,g,Y),(i_s,fk_s);r)$
$k \leftarrow\$ \{0,1\}^{\mathsf{RKE.kl}}$ $fk \leftarrow\$ \{0,1\}^{\mathsf{F.kl}}$ $hk \leftarrow\$ \{0,1\}^{\mathsf{H.kl}}$ $g \leftarrow\$ \mathbb{G}^*$; $y \leftarrow\$ \mathbb{Z}_p$ $stk_s \leftarrow (hk,g,g^y)$; $stk_r \leftarrow y$ $sek_s \leftarrow (0,fk)$ $sek_r \leftarrow (0,fk)$ $z \leftarrow (k,sek_s,(stk_s,stk_r,sek_r))$ Return z	$x \leftarrow r$; $X \leftarrow g^x$; $\sigma \leftarrow \mathsf{F.Ev}(fk_s,X)$ $s \leftarrow \mathsf{H.Ev}(hk,i_s \,\|\, \sigma \,\|\, X \,\|\, Y^x)$; $k_s \leftarrow s[1\ldots\mathsf{RKE.kl}]$ $fk_s \leftarrow s[\mathsf{RKE.kl}+1\ldots\mathsf{RKE.kl}+\mathsf{F.kl}]$ Return $((i_s+1,fk_s),k_s,(X,\sigma))$ Algorithm $\mathsf{RKE.RKg}((hk,g,Y),y,(i_r,fk_r),(X,\sigma),k_r)$ $acc \leftarrow (\sigma = \mathsf{F.Ev}(fk_r,X))$ If not acc then return $((i_r,fk_r),k_r,acc)$ $s \leftarrow \mathsf{H.Ev}(hk,i_r \,\|\, \sigma \,\|\, X \,\|\, X^y)$; $k_r \leftarrow s[1\ldots\mathsf{RKE.kl}]$ $fk_r \leftarrow s[\mathsf{RKE.kl}+1\ldots\mathsf{RKE.kl}+\mathsf{F.kl}]$ Return $((i_r+1,fk_r),k_r,acc)$

Figure 1.6. Ratcheted key exchange scheme $\mathsf{RKE} = \mathsf{RATCHET\text{-}KE}[\mathbb{G},\mathsf{F},\mathsf{H}]$.

Adversary $\mathcal{D}_1(stk_s)$	Adversary $\mathcal{D}_3(stk_s)$
$(hk,g,Y) \leftarrow stk_s$ $x \leftarrow\$ \mathbb{Z}_p$; $\textsc{RatRec}(g^x)$ $k_r \leftarrow \textsc{ChRec}$ $k_r' \leftarrow \mathsf{H.Ev}(hk,Y^x)$ If $k_r' = k_r$ then return 1 Else return 0	$upd_0 \leftarrow \textsc{RatSend}$; $\textsc{RatRec}(upd_0)$; $k_s \leftarrow \textsc{ChSend}$ $upd_1 \leftarrow \textsc{RatSend}$; $\textsc{RatRec}(upd_1)$ $(r,fk_s,k_s) \leftarrow \textsc{Exp}$; $(X_0,\sigma_0) \leftarrow upd_0$ $\sigma \leftarrow \mathsf{F.Ev}(fk_s,X_0)$; $upd_2 \leftarrow (X_0,\sigma)$ $\textsc{RatRec}(upd_2)$; $k_r \leftarrow \textsc{ChRec}$ If $k_s = k_r$ then return 1 else return 0
Adversary $\mathcal{D}_2(stk_s)$	**Adversary $\mathcal{D}_4(stk_s)$**
$(hk,g,Y) \leftarrow stk_s$ $x \leftarrow\$ \mathbb{Z}_p$; $\textsc{RatRec}(g^x)$ $k_r \leftarrow \textsc{ChRec}$ $(r,fk_s,k_s) \leftarrow \textsc{Exp}$ $k \,\|\, fk \leftarrow \mathsf{H.Ev}(hk,fk_s \,\|\, Y^x)$ If $k = k_r$ then return 1 Else return 0	$(r,sek_s,k_s) \leftarrow \textsc{Exp}$ $((i_s^*,fk_s^*),k_s^*,upd^*) \leftarrow\$ \mathsf{RKE.SKg}(stk_s,sek_s)$ $upd_0 \leftarrow \textsc{RatSend}$; $\textsc{RatRec}(upd^*)$ $upd_1 \leftarrow \textsc{RatSend}$; $(X,\sigma) \leftarrow upd_1$ $\sigma^* \leftarrow \mathsf{F.Ev}(fk_s^*,X)$; $upd^* \leftarrow (X,\sigma^*)$; $\textsc{RatRec}(upd^*)$ $k_s \leftarrow \textsc{ChSend}$; $k_r \leftarrow \textsc{ChRec}$ If $k_s = k_r$ then return 1 else return 0

Figure 1.7. Attacks against insecure variants of $\mathsf{RKE} = \mathsf{RATCHET\text{-}KE}[\mathbb{G},\mathsf{F},\mathsf{H}]$.

Design considerations.

We will examine some of the design decisions of RKE by considering several ratcheted key exchange schemes that are weakened versions of RKE, and corresponding adversaries that are able to successfully attack these schemes. The first two will omit the use of a MAC and thus be vulnerable to attacks where the adversary sends its own update information to $\textsc{RatRec}$ without having called $\textsc{Exp}$ first (though the second will have to make an expose query afterwards). In the latter two examples we consider variations of RKE that use fewer inputs to the hash function.

Our adversaries against these schemes thereby justify the choices we made for the input to the hash function. For the sake of compactness we omit showing that the constructed KIND adversaries have access to oracles $\text{RatSend}, \text{RatRec}, \text{Exp}, \text{ChSend}, \text{ChRec}$, and we omit showing that oracle calls return any output whenever this output is not used by the adversary.

Schemes without a MAC.

First let us consider changing RKE to not use its MAC F and instead simply use an unauthenticated g^x as its update information. For simplicity we will additionally assume that the only input to H is a group element g^{xy}. Consider adversary $\mathcal{D}_1$ shown in Fig. 1.7. It makes a RatRec query with a g^x of its own choice, then calls oracle ChRec and checks whether the key it received was real or random by comparing it to $\mathsf{H}(hk, Y^x)$. Referring to this weakened scheme as RKE_1, it is clear that $\mathsf{Adv}^{\mathsf{kind}}_{\mathsf{RKE}_1,\mathcal{D}_1} = 1 - 2^{-\mathsf{RKE.kl}}$.

Besides using a MAC, another way to prevent the specific attack given above would be to put a shared secret key fk into the hash function along with g^{xy} for every update. Let RKE_2 denote a version of RKE that still does not use a MAC but updates its keys with the hash function via $k \| fk \leftarrow \mathsf{H.Ev}(hk, fk \| g^{xy})$. An adversary like $\mathcal{D}_1$ will not work against RKE_2 because computing the new value of k requires knowing the secret value fk. But there is still a simple attack against RKE_2. Consider adversary $\mathcal{D}_2$ shown in Fig. 1.7. It works in the same way as $\mathcal{D}_1$ except it needs to make an expose query to obtain fk_s before it can compute k using the hash function. One subtle point to notice is that it is important that $\mathcal{D}_2$ calls Exp *after* its call to RatRec. Otherwise the $\mathsf{restricted}$ flag in KIND would have been set to true and ChRec would always return the real key (instead of returning a randomly chosen key when the challenge bit in KIND is set to 0). Having noticed this it is clear that $\mathsf{Adv}^{\mathsf{kind}}_{\mathsf{RKE}_2,\mathcal{D}_2} = 1 - 2^{-\mathsf{RKE.kl}}$.

In Section 1.6 we give an attack against *any* ratcheted encryption scheme, showing that if it is possible for an adversary to generate its own upd that the receiver will accept, than the adversary can use this ability to successfully attack the ratcheted encryption scheme. This proves that some sort of authentication is required for the update information if we want a scheme to be

secure.

Authenticating the update information in the Double Ratchet algorithm.

The default version of the Double Ratchet algorithm [48, 36] — which is used in the Signal protocol [54] — does not authenticate the update information. A single, one-sided version of this algorithm would evolve its keys in a way that is vaguely similar to the RKE_2 scheme discussed above, so it would not meet our security definition. This does not immediately lead to any real-world attacks, and could mean that our security definition is stronger than necessary. Furthermore, [36] describes the header encryption variant of the Double Ratchet algorithm. A single, one-sided version of this algorithm provides some form of authentication for update information and might meet our security definition.

Necessity of inputs to H.

In the construction of $\mathsf{RATCHET\text{-}KE}$, function $\mathsf{H}(hk,\cdot)$ takes a string $w = i \,\|\, \sigma_i \,\|\, g^{x_i} \,\|\, g^{x_i y}$ as input. The most straightforward part of w is $g^{x_i y}$, which provides unpredictability to ensure that the generated keys are indistinguishable from uniformly random strings. String w also includes the counter i, and the corresponding update information $upd_i = (g^{x_i}, \sigma_i)$. The inclusion of counter i in w ensures that an attacker cannot perform a "key-reuse" attack to make the receiver generate an output key that was already used before; we provide an example of such attack below. We also describe a "key-collision" attack against the KIND security of the scheme that is prevented by including upd_i in w. Finally, note that our concatenation operator $\|$ is defined to produce uniquely decodable strings, so the mapping of $(i, \sigma_i, g^{x_i}, g^{x_i y})$ into string w is injective; this helps to avoid attacks that take advantage of malleable encodings.

Key-reuse attack.

Game KIND makes sure that if challenge keys are acquired from the sender and the receiver for the same value of i (i.e. $i_s = i_r$), then these keys are consistent even if they are picked randomly. Otherwise it would be trivial to attack any ratcheted key exchange scheme. However, the game does not maintain such consistency between different values of i. Let RKE_3 denote

RKE if it was changed to use only g^{xy} as input to the hash function. Consider the "key-reuse" attack $\mathcal{D}_3$ shown in Fig. 1.7 that exploits the above as follows. Adversary $\mathcal{D}_3$ starts by calling RATSEND, RATREC and CHSEND to get a sender's challenge key k_s. Note that if the challenge bit is $b = 1$ in game KIND, then k_s equals to $\mathsf{H.Ev}(hk, Y^x)$ for some exponent x generated during RATSEND. Next, the adversary calls both RATSEND and RATREC to ratchet the key forward, in order to be able to make EXP queries. It calls EXP to get fk_s so that it can re-authenticate the same value of $X = g^x$ that was used for the sender's challenge query. Then it sends X and its new MAC tag σ to the receiver, which sets the restricted flag true. The latter means that calling CHREC results in getting the receiver's real output key regardless of the challenge bit. If this key is equal to the previously learned sender's challenge key then it is highly likely that the challenge bit b equals 1, otherwise it must be 0. This gives the advantage of $\mathsf{Adv}^{\mathsf{kind}}_{\mathsf{RKE}_3,\mathcal{D}_3} = 1 - 2^{-\mathsf{RKE.kl}}$.

Key-collision attacks.

We now describe the final attack idea that does not work against our construction but would have been possible if the update information $upd = (g^{x_i}, \sigma)$ was not included in the hash function. Consider changing RATCHET-KE[$\mathbb{G}$, F, H] to have $\mathsf{H}(hk, \cdot)$ take inputs of the form $w = i \| g^{x_i y}$. Call this scheme RKE_4. This enables the following attack, as defined by the adversary $\mathcal{D}_4$ in Fig. 1.7. Assume that an attacker compromises the sender's keys k_s and fk_s and immediately uses the compromised authenticity to establish new keys k_s^* and fk_s^*, shared between the attacker and the receiver. Now let $upd = (X, \sigma)$ be the next update information produced by the sender. The attacker can construct malicious update information $upd^* = (X, \sigma^*)$, where $\sigma^* = \mathsf{F.Ev}(fk_s^*, X)$, and send it to the receiver. The receiver would accept upd^* and use the output of $\mathsf{H.Ev}(hk, i \| X^y)$ as new key material, resulting in the same keys as those generated by the sender. Now the the receiver and the sender share an output key, while the restricted flag is set true, so checking whether the output of the two challenge oracles is the same yields a good attack.

We will not give the exact advantage of $\mathcal{D}_4$. If σ^* and σ happen to be exactly the same,

then the restricted flag would be set back to false and the attack would fail because the two keys received from the sender's and the reciever's challenge oracles would be the same regardless of game's challenge bit. But if $\sigma^* = \sigma$ was likely to occur then the ratcheted key exchange scheme would be insecure for other reasons. One could formalize this by building a second adversary against RKE_4 to show that one of the two adversaries must have a high advantage. For the purpose of this section we simply note that this event is extremely unlikely to occur for any typical choice of hash function and MAC.

1.3.4 Security proof for our ratcheted key exchange scheme

In previous section we showed that several variations of our ratcheted key exchange scheme $\mathsf{RKE} = \mathsf{RATCHET\text{-}KE}[\mathbb{G}, \mathsf{F}, \mathsf{H}]$ are insecure. In this section we will prove that our scheme is secure. We now present our theorem bounding the advantage of an adversary breaking the KIND-security of RKE by that of adversaries against the SUFCMA-security of F and the ODHE-security of $\mathbb{G}, \mathsf{H}$.

Theorem 1. *Let $\mathbb{G}$ be a cyclic group of order $p \in \mathbb{N}$, and let $\mathbb{G}^*$ denote the set of its generators. Let F be a function family such that $\mathsf{F.In} = \mathbb{G}$. Let H be a function family such that $\mathsf{H.In} = \{0,1\}^*$ and $\mathsf{H.ol} > \mathsf{F.kl}$. Let $\mathsf{RKE} = \mathsf{RATCHET\text{-}KE}[\mathbb{G}, \mathsf{F}, \mathsf{H}]$. Let $\mathcal{D}$ be an adversary attacking the* KIND-*security of* RKE *that makes q_{RatSend} queries to its* RatSend *oracle, q_{RatRec} queries to its* RatRec *oracle, q_{Exp} queries to its* Exp *oracle, q_{ChSend} queries to its* ChSend *oracle, and q_{ChRec} queries to its* ChRec *oracle. Then there is an adversary $\mathcal{F}$ attacking the* SUFCMA-*security of* F*, and adversaries $\mathcal{O}_1, \mathcal{O}_2$ attacking the* ODHE-*security of $\mathbb{G}, \mathsf{H}$, such that*

$$\mathsf{Adv}^{\mathsf{kind}}_{\mathsf{RKE},\mathcal{D}} \leq 2 \cdot (q_{\text{RatSend}} + 1) \cdot \mathsf{Adv}^{\mathsf{sufcma}}_{\mathsf{F},\mathcal{F}} + 2 \cdot q_{\text{RatSend}} \cdot \mathsf{Adv}^{\mathsf{odhe}}_{\mathbb{G},\mathsf{H},\mathcal{O}_1} + 2 \cdot \mathsf{Adv}^{\mathsf{odhe}}_{\mathbb{G},\mathsf{H},\mathcal{O}_2}.$$

Adversary $\mathcal{F}$ makes at most q_{RatSend} queries to its Tag *oracle and q_{RatRec} queries to its* Verify *oracle. Adversary $\mathcal{O}_1$ makes at most q_{RatSend} queries to its* Up *oracle,* 2 *queries to its* Ch *oracle, q_{Exp} queries to its* Exp *oracle, and $q_{\text{RatSend}} + q_{\text{RatRec}} - 2$ queries to its* Hash *oracle. Adversary $\mathcal{O}_2$ makes at most q_{RatSend} queries to its* Up *oracle, $q_{\text{RatSend}} + q_{\text{RatRec}}$*

queries to its CH *oracle,* q_{EXP} *queries to its* EXP *oracle, and* $q_{\text{RATREC}} + q_{\text{EXP}}$ *queries to its* HASH *oracle. Each of* $\mathcal{F}$, $\mathcal{O}_1$, $\mathcal{O}_2$ *has a running time approximately that of* $\mathcal{D}$.

The proof requires careful attention to detail due to subtleties. The most natural proof method may be to proceed one RATSEND query at a time, first replacing the output of the hash function with random bits (unless an expose happens) and then using the security of the MAC to argue that the adversary cannot produce any modified update information that will be accepted by the receiver without exposing. But there is a subtle flaw with this proof technique. The adversary may attempt to create a forged upd before it has decided whether to expose. In this case we need to check the validity of their forgery with a MAC key, before we know whether it should be random or a valid output of the hash function.

To avoid this problem we first use a hybrid argument to show that no such forgery is possible before replacing all non-exposed keys with random. We proceed one RATSEND query at a time, showing that we can temporarily replace the key with random when checking the sort of attempted forgery described above. This then allows us to use the security of the MAC to assume that the forgery attempt failed without us having to commit to a key to verify with. We thus are able to show one step at a time that all such forgery attempts can be assumed to fail without having to check.

Once this is done, we are never forced to use a key before the adversary has committed to whether it will perform a relevant exposure of the secret state. As such we can safely delay our decision of whether or not the key should be replaced by random values until it is known whether an expose will happen. This allows us to use the ODHE security of H and $\mathbb{G}$ to argue that we can replace all of the generated keys with randomness, only using H to generate the real keys at the last moment whenever an expose query is made.

Theorem 1. Consider the sequence of games shown in Fig. 1.8. Lines not annotated with comments are common to all games. $\text{G}_{0,0}$ is identical to $\text{KIND}^{\mathcal{D}}_{\mathsf{RKE}}$ with the code of RKE inserted. Additionally, a flag $\mathsf{unchanged}$ has been added. This flag keeps track of whether the

most recent update information was passed unchanged from the sender to the receiver and thus the keys k_r and fk_r should be indistinguishable from random to adversary $\mathcal{D}$. In this case, the adversary should not be able to create update information upd that is accepted by $\textsc{RatRec}$ unless it calls $\textsc{Exp}$ or forwards along the upd generated by the sender. We prove this with a hybrid argument over the games $\mathrm{G}_{0,0},\ldots,\mathrm{G}_{0,q_{\textsc{RatSend}}+1}$. Game $\mathrm{G}_{0,j}$ assumes forgery attempts fail for the first j keys, sets a bad flag if $\mathcal{D}$ is successful at forging against the $(j+1)$-th key, and performs normally for all following keys. Game $\mathrm{G}^*_{0,j}$ is the same except it also acts as if $\mathcal{D}$ failed to forge even when the bad flag is set. Thus, from the perpective of an adversary $\mathrm{G}^*_{0,j}$ is simply assumping that forgery attempts fail for the first $j+1$ keys, making it equivalent to $\mathrm{G}_{0,j+1}$. Thus for all $j \in \{0,\ldots,q_{\textsc{RatSend}}\}$,

$$\Pr[\mathrm{G}_{0,0}] = \Pr[\mathrm{KIND}^{\mathcal{D}}_{\mathsf{RKE}}] \quad \text{and} \quad \Pr[\mathrm{G}^*_{0,j}] = \Pr[\mathrm{G}_{0,j+1}].$$

Furthermore, for all $j \in \{1,\ldots,q_{\textsc{RatSend}}\}$, games $\mathrm{G}_{0,j}$ and $\mathrm{G}^*_{0,j}$ are identical until bad, so the fundamental lemma of game playing [16] gives:

$$\Pr[\mathrm{G}_{0,j}] - \Pr[\mathrm{G}^*_{0,j}] \leq \Pr[\mathsf{bad}^{\mathrm{G}^*_{0,j}}],$$

where $\Pr[\mathsf{bad}^{\mathsf{Q}}]$ denotes the probability of setting the bad flag in game Q.

We cannot directly bound $\Pr[\mathsf{bad}^{\mathrm{G}^*_{0,j}}]$ using the security of F because the key being used for F is chosen as output from H instead of uniformly at random, consider the relationship between games $\mathrm{G}^*_{0,j}$ and I_j (the latter also shown in Fig. 1.8). Game I_j is identical to $\mathrm{G}^*_{0,j}$, except that in I_j the output of hash function H is replaced with a uniformly random string whenever $i+1=j$ (thus the key used to check whether bad should be set when $i=j$ is uniformly random).

Note that when $j=0$ the games $\mathrm{G}^*_{0,0}$ and I_0 are identical so $\Pr[\mathsf{bad}^{\mathrm{G}^*_{0,0}}] = \Pr[\mathsf{bad}^{\mathrm{I}_0}]$. For other values of j we relate the probability that these games set bad to the advantage of the oracle Diffie-Hellman adversary $\mathcal{O}_1$ that is defined in Fig. 1.10. Adversary $\mathcal{O}_1$ picks j' at random and then uses its oracles to simulate $\mathrm{G}^*_{0,j}$ or I_j. Then if the bad flag is set it sets a bit b' equal to 1. This bit is ultimately returned by $\mathcal{O}$. Thus the probability that $\mathcal{O}$ outputs 1 is exactly the probability that the bad flag would be set in the game it is simulating.

Games $\mathrm{G}_{0,j}$, $\mathrm{G}^*_{0,j}$, I_j

$b \leftarrow\$ \{0,1\}$; $i_s \leftarrow 0$; $i_r \leftarrow 0$; unchanged $\leftarrow$ true ; $rand \leftarrow\$ \{0,1\}^{\mathsf{H.ol}}$
$k_s \leftarrow\$ \{0,1\}^{\mathsf{RKE.kl}}$; $k_r \leftarrow k_s$; $fk_s \leftarrow\$ \{0,1\}^{\mathsf{F.kl}}$; $fk_r \leftarrow fk_s$
$hk \leftarrow\$ \{0,1\}^{\mathsf{H.kl}}$; $g \leftarrow\$ \mathbb{G}^*$; $y \leftarrow\$ \mathbb{Z}_p$; $stk_s \leftarrow (hk, g, g^y)$
$b' \leftarrow\$ \mathcal{D}^{\textsc{RatSend},\textsc{RatRec},\textsc{Exp},\textsc{ChSend},\textsc{ChRec}}(stk_s)$; Return $(b' = b)$

RatSend

If op$[i_s] = \bot$ then op$[i_s] \leftarrow$ "ch"
$x \leftarrow\$ \mathbb{Z}_p$; $\sigma \leftarrow \mathsf{F.Ev}(fk_s, g^x)$; $upd \leftarrow (g^x, \sigma)$
$s \leftarrow \mathsf{H.Ev}(hk, i_s \| \sigma \| g^x \| g^{xy})$
If $i_s + 1 = j$ then $s \leftarrow rand$ // I_j
auth$[i_s] \leftarrow upd$; $i_s \leftarrow i_s + 1$; $k_s \leftarrow s[1 \ldots \mathsf{RKE.kl}]$
$fk_s \leftarrow s[\mathsf{RKE.kl} + 1 \ldots \mathsf{RKE.kl} + \mathsf{F.kl}]$; Return upd

RatRec(upd)

$(X, \sigma) \leftarrow upd$
If unchanged and (op$[i_r] \neq$ "exp") and ($upd \neq$ auth$[i_r]$) then
  If $i_r < j$ then return false
  If $i_r = j$ then
    If $\sigma \neq \mathsf{F.Ev}(fk_r, X)$ then return false
    bad $\leftarrow$ true
    Return false // $\mathrm{G}^*_{0,j}$, I_j
If $\sigma \neq \mathsf{F.Ev}(fk_r, X)$ then return false
If op$[i_r]$ = "exp" then restricted $\leftarrow$ true
If upd = auth$[i_r]$ then
  unchanged $\leftarrow$ true ; restricted $\leftarrow$ false
Else
  unchanged $\leftarrow$ false
$s \leftarrow \mathsf{H.Ev}(hk, i_r \| \sigma \| X \| X^y)$
If $i_r + 1 = j$ then $s \leftarrow rand$ // I_j
$i_r \leftarrow i_r + 1$; $k_r \leftarrow s[1 \ldots \mathsf{RKE.kl}]$
$fk_r \leftarrow s[\mathsf{RKE.kl} + 1 \ldots \mathsf{RKE.kl} + \mathsf{F.kl}]$; Return true

Exp

If op$[i_s]$ = "ch" then return $\bot$
op$[i_s] \leftarrow$ "exp" ; Return $(x, (i_s, fk_s), k_s)$

ChSend

//Unchanged from KIND

ChRec

//Unchanged from KIND

Figure 1.8. Games $\mathrm{G}_{0,j}, \mathrm{G}^*_{0,j}, \mathrm{I}_j$ for proof of Theorem 1.

Let b_{odhe} denote the challenge bit in game $\mathrm{ODHE}^{\mathcal{O}_1}_{\mathbb{G},\mathsf{H}}$, and let b' denote the corresponding guess made by the adversary $\mathcal{O}_1$. Let j' be the value sampled in the first step of $\mathcal{O}_1$. For each

```
Games G1–G2
b ←$ {0,1} ; i_s ← 0 ; i_r ← 0 ; unchanged ← true
k_s[0] ←$ {0,1}^RKE.kl ; k_r ← k_s[0] ; fk_s[0] ←$ {0,1}^F.kl ; fk_r ← fk_s[0]
hk ←$ {0,1}^H.kl ; g ←$ G* ; y ←$ Z_p ; stk_s ← (hk, g, g^y)
b' ←$ D^{RatSend,RatRec,Exp,ChSend,ChRec}(stk_s) ; Return (b' = b)

RatSend
If op[i_s] = ⊥ then op[i_s] ← "ch"
x ←$ Z_p ; σ ← F.Ev(fk_s[i_s], g^x) ; upd ← (g^x, σ)
s ← H.Ev(hk, i_s ∥ σ ∥ g^x ∥ g^xy)                          // G1
s ←$ {0,1}^H.ol                                              // G2
auth[i_s] ← upd ; i_s ← i_s + 1 ; k_s[i_s] ← s[1 ... RKE.kl]
fk_s[i_s] ← s[RKE.kl + 1 ... RKE.kl + F.kl] ; Return upd

RatRec(upd)
(X, σ) ← upd
If unchanged and (op[i_r] ≠ "exp") and (upd ≠ auth[i_r]) then
   Return false
If unchanged then fk_r ← fk_s[i_r]
If (σ ≠ F.Ev(fk_r, X)) then return false
If op[i_r] = "exp" then restricted ← true
If upd = auth[i_r]
   unchanged ← true ; restricted ← false ; i_r ← i_r + 1
Else
   unchanged ← false
   s ← H.Ev(hk, i_r ∥ σ ∥ X ∥ X^y)
   i_r ← i_r + 1 ; k_r ← s[1 ... RKE.kl]
   fk_r ← s[RKE.kl + 1 ... RKE.kl + F.kl]
Return true
```

```
Exp
If op[i_s] = "ch" then return ⊥
op[i_s] ← "exp" ; (X, σ) ← auth[i_s − 1]
s ← H.Ev(hk, (i_s − 1) ∥ σ ∥ X ∥ X^y)
k_s[i_s] ← s[1 ... RKE.kl]
fk_s[i_s] ← s[RKE.kl + 1 ... RKE.kl + F.kl]
Return (x, (i_s, fk_s[i_s]), k_s[i_s])

ChSend
If op[i_s] = "exp" then return ⊥
op[i_s] ← "ch"
If rkey[i_s] = ⊥ then rkey[i_s] ←$ {0,1}^RKE.kl
If b = 1 then return k_s[i_s] else return rkey[i_s]
```

```
ChRec
If restricted then return k_r
If op[i_r] = "exp" then return ⊥
op[i_r] ← "ch"
If rkey[i_r] = ⊥ then rkey[i_r] ←$ {0,1}^RKE.kl
If unchanged then k_r ← k_s[i_r]
If b = 1 then return k_r else return rkey[i_r]
```

Figure 1.9. Games G_1, G_2 for proof of Theorem 1.

choice of j', adversary $\mathcal{O}_1$ perfectly simulates the view of $\mathcal{D}$ in either $\mathrm{G}^*_{0,j'}$ or $\mathrm{I}_{j'}$ depending on whether its $\mathrm{C}\textsc{h}$ oracle is returning real output of the hash function or a random value. If $\mathcal{D}$ performs an action that would prevent bad from being set (such as calling $\mathrm{E}\textsc{xp}$ when $i_s = j'$) then $\mathcal{O}_1$ no longer perfectly simulates the view of $\mathcal{D}$, but it does not matter for our analysis because we already know bad (and thus b') will not be set. So for all $j \in \{1,\ldots,q_{\mathrm{RatSend}}\}$, we have

$$\Pr[\mathsf{bad}^{\mathrm{G}^*_{0,j}}] = \Pr[b' = 1 \,|\, b_{\mathsf{odhe}} = 1, j' = j],$$

$$\Pr[\mathsf{bad}^{\mathrm{I}_j}] = \Pr[b' = 1 \,|\, b_{\mathsf{odhe}} = 0, j' = j].$$

Combining the above for all values of j (using $\Pr[\mathsf{bad}^{\mathrm{G}^*_{0,0}}] = \Pr[\mathsf{bad}^{\mathrm{G}i_s}]$) gives

$$\begin{aligned}&\mathsf{Adv}^{\mathsf{odhe}}_{\mathbb{G},\mathsf{H},\mathcal{O}_1} = \Pr[b' = 1 \,|\, b_{\mathsf{odhe}} = 1] - \Pr[b' = 1 \,|\, b_{\mathsf{odhe}} = 0]\\ &= \sum_{j=1}^{q_{\mathrm{RatSend}}} \Pr[j = j'](\Pr[\mathsf{bad}^{\mathrm{G}^*_{0,j}}] - \Pr[\mathsf{bad}^{\mathrm{I}_j}]) = \sum_{j=0}^{q_{\mathrm{RatSend}}} \frac{\Pr[\mathsf{bad}^{\mathrm{G}^*_{0,j}}] - \Pr[\mathsf{bad}^{\mathrm{I}_j}]}{q_{\mathrm{RatSend}}}.\end{aligned}$$

Note that we were able to change the starting index of j for that last summation because $\Pr[\mathsf{bad}^{\mathrm{G}^*_{0,0}}] = \Pr[\mathsf{bad}^{\mathrm{I}_0}]$, as we noted before.

To complete the hybrid argument part of the proof, we can finally bound the probability that bad gets set true in I_j. Doing so requires adversary $\mathcal{D}$ to successfully forge a MAC tag for a uniformly random key, allowing us to reduce to the security of F. Formally, we use $\mathcal{D}$ to construct an adversary $\mathcal{F}$ attacking the SUFCMA security of F. Adversary $\mathcal{F}$ (shown in Fig. 1.11) simulates adversary $\mathcal{D}$ and guesses when it will first create a forgery. $\mathcal{F}$ simulates game I_j for adversary $\mathcal{D}$ until that point, and uses its own SUFCMA oracles to answer $\mathcal{D}$'s queries at the time when it expects the forgery. Similar to the earlier case when $\mathcal{O}_1$ simulated $\mathcal{D}$, adversary $\mathcal{F}$ may fail to simulate I_j for adversary $\mathcal{D}$ when the latter performs certain actions that preclude bad from being set. This does not affect our analysis because we only require that if bad is set then $\mathcal{F}$ will return a successful forgery.

Adversary $\mathcal{O}_1^{\textsc{Up},\textsc{Ch},\textsc{Exp},\textsc{Hash}}(hk,g,Y)$

$j' \leftarrow_{\$} \{1,\ldots,q_{\textsc{RatSend}}\}$; $b \leftarrow_{\$} \{0,1\}$; $b' \leftarrow 0$
$i_s \leftarrow 0$; $i_r \leftarrow 0$; unchanged $\leftarrow$ true
$k_s \leftarrow_{\$} \{0,1\}^{\mathsf{RKE.kl}}$; $k_r \leftarrow k_s$
$fk_s \leftarrow_{\$} \{0,1\}^{\mathsf{F.kl}}$; $fk_r \leftarrow fk_s$; $stk_s \leftarrow (hk,g,Y)$
$\mathcal{D}^{\textsc{RatSendSim},\textsc{RatRecSim},\textsc{ExpSim},\textsc{ChSendSim},\textsc{ChRecSim}}(stk_s)$
Return b'

$\textsc{RatRecSim}(upd)$

$(X,\sigma) \leftarrow upd$
forge $\leftarrow ((\mathsf{op}[i_r] \neq \text{“exp”}) \wedge (upd \neq \mathsf{auth}[i_r]))$
If unchanged and forge then
  If $i_r < j'$ then return false
  If $i_r = j'$ then
    If $\sigma \neq \mathsf{F.Ev}(fk_r,X)$ then return false
    bad $\leftarrow$ true ; $b' \leftarrow 1$; Return false
If $\sigma \neq \mathsf{F.Ev}(fk_r,X)$ then return false
If $\mathsf{op}[i_r] =$ “exp” then restricted $\leftarrow$ true
If $upd = \mathsf{auth}[i_r]$ then
  unchanged $\leftarrow$ true ; restricted $\leftarrow$ false
Else unchanged $\leftarrow$ false
If $i_r + 1 \neq j'$ then $s \leftarrow \textsc{Hash}(i_r, \sigma \| X, X)$
Else $s \leftarrow \textsc{Ch}(\sigma \| X)$
$i_r \leftarrow i_r + 1$; $k_r \leftarrow s[1 \ldots \mathsf{RKE.kl}]$
$fk_r \leftarrow s[\mathsf{RKE.kl}+1 \ldots \mathsf{RKE.kl}+\mathsf{F.kl}]$
Return true

$\textsc{ExpSim}$

If $\mathsf{op}[i_s] =$ “ch” then return $\perp$
$\mathsf{op}[i_s] \leftarrow$ “exp” ; $x \leftarrow \textsc{Exp}$
Return $(x,(i_s,fk_s),k_s)$

$\textsc{RatSendSim}$

If $\mathsf{op}[i_s] = \perp$ then $\mathsf{op}[i_s] \leftarrow$ “ch”
$X \leftarrow \textsc{Up}$; $\sigma \leftarrow \mathsf{F.Ev}(fk_s,X)$
$upd \leftarrow (X,\sigma)$
If $i_s + 1 \neq j'$ then
  $s \leftarrow \textsc{Hash}(i_s, \sigma \| X, X)$
Else
  $s \leftarrow \textsc{Ch}(\sigma \| X)$
$\mathsf{auth}[i_s] \leftarrow upd$; $i_s \leftarrow i_s + 1$
$k_s \leftarrow s[1 \ldots \mathsf{RKE.kl}]$
$fk_s \leftarrow s[\mathsf{RKE.kl}+1 \ldots \mathsf{RKE.kl}+\mathsf{F.kl}]$
Return upd

$\textsc{ChSendSim}$

If $\mathsf{op}[i_s] =$ “exp” then return $\perp$
$\mathsf{op}[i_s] \leftarrow$ “ch”
If $\mathsf{rkey}[i_s] = \perp$ then
  $\mathsf{rkey}[i_s] \leftarrow_{\$} \{0,1\}^{\mathsf{RKE.kl}}$
If $b = 1$ then return k_s
Else return $\mathsf{rkey}[i_s]$

$\textsc{ChRecSim}$

If restricted then return k_r
If $\mathsf{op}[i_r] =$ “exp” then return $\perp$
$\mathsf{op}[i_r] \leftarrow$ “ch”
If $\mathsf{rkey}[i_r] = \perp$ then
  $\mathsf{rkey}[i_r] \leftarrow_{\$} \{0,1\}^{\mathsf{RKE.kl}}$
If $b = 1$ then return k_r
Else return $\mathsf{rkey}[i_r]$

Figure 1.10. Adversary $\mathcal{O}_1$ for proof of Theorem 1.

Thus for $j \in \{0,\ldots,q_{\textsc{RatSend}}\}$, we have $\Pr[\mathsf{bad}^{\mathrm{I}_j}] \leq \Pr[\,\mathrm{SUFCMA}_{\mathsf{F}}^{\mathcal{F}} \mid j' = j\,]$ which gives $\mathsf{Adv}^{\mathsf{sufcma}}_{\mathsf{F},\mathcal{F}} \geq (1/(q_{\textsc{RatRec}}+1)) \sum_{j=0}^{q_{\textsc{RatRec}}} \Pr[\mathsf{bad}^{\mathrm{I}_j}]$.

The above work allows us to transition to game $\mathrm{G}_{0,q_{\textsc{RatSend}}+1}$ as shown in the following equations. From there we will move to games $\mathrm{G}_1,\mathrm{G}_2$ shown in Fig. 1.9. All of the summations

Adversary $\mathcal{F}^{\text{TAG},\text{VERIFY}}$

$j' \leftarrow\$ \{0,\ldots,q_{\text{RATSEND}}\}\,;\ b \leftarrow\$ \{0,1\}$
$i_s \leftarrow 0\,;\ i_r \leftarrow 0\,;\ \mathsf{unchanged} \leftarrow \mathsf{true}$
$rand \leftarrow\$ \{0,1\}^{\mathsf{H.ol}}\,;\ k_s \leftarrow\$ \{0,1\}^{\mathsf{RKE.kl}}\,;\ k_r \leftarrow k_s$
$fk_s \leftarrow\$ \{0,1\}^{\mathsf{F.kl}}\,;\ fk_r \leftarrow fk_s\,;\ hk \leftarrow\$ \{0,1\}^{\mathsf{H.kl}}$
$g \leftarrow\$ \mathbb{G}^*\,;\ y \leftarrow\$ \mathbb{Z}_p\,;\ stk_s \leftarrow (hk, g, g^y)$
$\mathcal{D}^{\text{RATSENDSIM},\text{RATRECSIM},\text{EXPSIM},\text{CHSENDSIM},\text{CHRECSIM}}(stk_s)$

RATRECSIM(upd)

$(X,\sigma) \leftarrow upd$
$\mathsf{forge} \leftarrow ((\mathsf{op}[i_r] \neq \text{“exp”}) \wedge (upd \neq \mathsf{auth}[i_r]))$
If unchanged and forge then
  If $i_r < j'$ then return false
  If $i_r = j'$ then
    If not $\text{VERIFY}(X,\sigma)$ then return false
    bad $\leftarrow$ true
    Return false
If $(i_r = j')$ then
  If not $\text{VERIFY}(X,\sigma)$ then return false
Else
  If $\sigma \neq \mathsf{F.Ev}(fk_r, X)$ then return false
If $\mathsf{op}[i_r] =$ “exp” then restricted $\leftarrow$ true
If $upd = \mathsf{auth}[i_r]$ then
  unchanged $\leftarrow$ true; restricted $\leftarrow$ false
Else
  unchanged $\leftarrow$ false
$s \leftarrow \mathsf{H.Ev}(hk, i_r \,\|\, \sigma \,\|\, X \,\|\, X^y)$
If $i_r + 1 = j$ then $s \leftarrow rand$
$i_r \leftarrow i_r + 1\,;\ k_r \leftarrow s[1\ldots\mathsf{RKE.kl}]$
$fk_r \leftarrow s[\mathsf{RKE.kl}+1\ldots\mathsf{RKE.kl}+\mathsf{F.kl}]$
Return true

RATSENDSIM

If $\mathsf{op}[i_s] = \bot$ then $\mathsf{op}[i_s] \leftarrow$ “ch”
$x \leftarrow\$ \mathbb{Z}_p$
If $i_s = j'$ then $\sigma \leftarrow \text{TAG}(g^x)$
Else $\sigma \leftarrow \mathsf{F.Ev}(fk_s, g^x)$
$s \leftarrow \mathsf{H.Ev}(hk, i_s \,\|\, \sigma \,\|\, g^x \,\|\, g^{xy})$
If $i_s + 1 = j$ then $s \leftarrow rand$
$upd \leftarrow (g^x, \sigma)\,;\ \mathsf{auth}[i_s] \leftarrow upd$
$i_s \leftarrow i_s + 1\,;\ k_s \leftarrow s[1\ldots\mathsf{RKE.kl}]$
$fk_s \leftarrow s[\mathsf{RKE.kl}+1\ldots\mathsf{RKE.kl}+\mathsf{F.kl}]$
Return upd

EXPSIM

If $\mathsf{op}[i_s] =$ “ch” then return $\bot$
$\mathsf{op}[i_s] \leftarrow$ “exp”
Return $(x, (i_s, fk_s), k_s)$

CHSENDSIM

If $\mathsf{op}[i_s] =$ “exp” then return $\bot$
$\mathsf{op}[i_s] \leftarrow$ “ch”
If $\mathsf{rkey}[i_s] = \bot$ then
  $\mathsf{rkey}[i_s] \leftarrow\$ \{0,1\}^{\mathsf{RKE.kl}}$
If $b = 1$ then return k_s
Else return $\mathsf{rkey}[i_s]$

CHRECSIM

If restricted then return k_r
If $\mathsf{op}[i_r] =$ “exp” then return $\bot$
$\mathsf{op}[i_r] \leftarrow$ “ch”
If $\mathsf{rkey}[i_r] = \bot$ then
  $\mathsf{rkey}[i_r] \leftarrow\$ \{0,1\}^{\mathsf{RKE.kl}}$
If $b = 1$ then return k_r
Else return $\mathsf{rkey}[i_r]$

Figure 1.11. Adversary $\mathcal{F}$ for proof of Theorem 1.

---

below are from $j = 0$ to $j = q_{\text{RATSEND}}$.

$$\Pr[\mathrm{KIND}^{\mathcal{D}}_{\mathsf{RKE}}] = \Pr[\mathrm{G}_{0,0}] = \Pr[\mathrm{G}_{1,q_{\text{RATSEND}}}] + \textstyle\sum_j \Pr[\mathrm{G}_{0,j}] - \Pr[\mathrm{G}^*_{0,j}]$$

$$\leq \Pr[\mathrm{G}_{1,q_{\text{RATSEND}}}] + \textstyle\sum_j \Pr[\mathsf{bad}^{\mathrm{G}^*_{0,j}}]$$

$$= \Pr[\mathrm{G}_{1,q_{\text{RATSEND}}}] + q_{\text{RATSEND}} \cdot \mathsf{Adv}^{\mathsf{odhe}}_{\mathbb{G},\mathsf{H},\mathcal{O}_1} + \textstyle\sum_j \Pr[\mathsf{bad}^{\mathrm{I}_j}]$$

$$\leq q_{\text{RATSEND}} \cdot \mathsf{Adv}^{\mathsf{odhe}}_{\mathbb{G},\mathsf{H},\mathcal{O}_1} + (q_{\text{RATSEND}} + 1) \cdot \mathsf{Adv}^{\mathsf{sufcma}}_{\mathsf{F},\mathcal{F}} + \Pr[\mathrm{G}_{1,q_{\text{RATSEND}}}].$$

```
Adversary O_2^{Up,Ch,Exp,Hash}(hk, g, Y)
b ←$ {0,1} ; i_s ← 0 ; i_r ← 0 ; unchanged ← true
k_s[0] ←$ {0,1}^{RKE.kl} ; k_r ← k_s[0]
fk_s[0] ←$ {0,1}^{F.kl} ; fk_r ← fk_s[0] ; hk ←$ {0,1}^{H.kl}
g ←$ G* ; y ←$ Z_p ; stk_s ← (hk, g, Y)
b' ←$ D^{RatSendSim,RatRecSim,ExpSim,ChSendSim,ChRecSim}(stk_s)
If (b' = b) then return 1 else return 0

RatSendSim
If op[i_s] = ⊥ then
   op[i_s] ← "ch"
   If i_s ≠ 0 then
      (X, σ) ← auth[i_s − 1] ; s ← Ch(σ||X)
      SaveKeys(i_s, s)
X ← Up ; σ ← F.Ev(fk_s[i_s], X) ; upd ← (X, σ)
auth[i_s] ← upd ; i_s ← i_s + 1 ; Return upd

RatRecSim(upd)
(X, σ) ← upd
forge ← ((op[i_r] ≠ "exp") ∧ (upd ≠ auth[i_r]))
If unchanged and forge then return false
If unchanged then fk_r ← fk_s[i_r]
If (σ ≠ F.Ev(fk_r, X)) then return false
If op[i_r] = "exp" then restricted ← true
If upd = auth[i_r]
   unchanged ← true ; restricted ← false ; i_r ← i_r + 1
Else
   unchanged ← false ; s ← Hash(i_r, σ||X, X)
   i_r ← i_r + 1 ; k_r ← s[1...RKE.kl]
   fk_r ← s[RKE.kl + 1...RKE.kl + F.kl]
Return true

SaveKeys(i, s)
k_s[i] ← s[1...RKE.kl]
fk_s[i] ← s[RKE.kl + 1...RKE.kl + F.kl]
```

```
ExpSim
If op[i_s] = "ch" then return ⊥
If (op[i_s] = ⊥) and (i_s ≠ 0) then
   x ← Exp
   (X, σ) ← auth[i_s − 1]
   s ← Hash(i_s − 1, σ||X, X)
   SaveKeys(i_s, s)
op[i_s] ← "exp"
Return (x, (i_s, fk_s[i_s]), k_s[i_s])

ChSendSim
If op[i_s] = "exp" then return ⊥
If (op[i_s] = ⊥) and (i_s ≠ 0) then
   (X, σ) ← auth[i_s − 1]
   s ← Ch(σ||X)
   SaveKeys(i_s, s)
op[i_s] ← "ch"
If rkey[i_s] = ⊥ then
   rkey[i_s] ←$ {0,1}^{RKE.kl}
If b = 1 then return k_s[i_s]
Else return rkey[i_s]

ChRecSim
If restricted then return k_r
If op[i_r] = "exp" then return ⊥
If (op[i_r] = ⊥) and (i_r ≠ 0) then
   (X, σ) ← auth[i_r − 1]
   s ← Ch(σ||X)
   SaveKeys(i_r, s)
op[i_r] ← "ch"
If rkey[i_r] = ⊥ then
   rkey[i_r] ←$ {0,1}^{RKE.kl}
If unchanged then k_r ← k_s[i_r]
If b = 1 then return k_r
Else return rkey[i_r]
```

Figure 1.12. Adversary $\mathcal{O}_2$ for proof of Theorem 1.

Game G_1 is identical to $G_{0,q_{\text{RatSend}}+1}$, but has been rewritten to allow make the final game transition of our proof easier to follow. The complicated, nested if-condition at the beginning of RatRec has been simplified because $i_r < q_{\text{RatSend}} + 1$ always holds when unchanged is true. Additionally, when unchanged is true (and thus upd has been directly forwarded between RatSend and RatRec without being modified) we delay setting k_r, fk_r until they are about

to be used, at which point they are set to match the appropriate k_s, fk_s that have been stored in a table. We have $\Pr[G_{0,q_{\text{RATSEND}}+1}] = \Pr[G_1]$.

Games G_1 and G_2 differ only in that, in G_2, values of k_0 and fk_s are chosen at random instead of as the output of H (unless EXP is called in which case we reset them to the correct output of H). We bound the difference between $\Pr[G_1]$ and $\Pr[G_2]$ by the advantage of the Diffie-Hellman adversary $\mathcal{O}_2$ that is defined in Fig. 1.12.

Let b_{odhe} denote the challenge bit in game $\text{ODHE}_{\mathbb{G},\mathsf{H}}^{\mathcal{O}_2}$, and let b' denote the corresponding guess made by the adversary $\mathcal{O}_2$. $\mathcal{O}_2$ uses its own oracle to simulate the view of $\mathcal{D}$. When $b_{\mathsf{odhe}} = 1$ it perfectly simulates the view of $\mathcal{D}$ in G_1, and when $b_{\mathsf{odhe}} = 0$ it perfectly simulates the view of G_2. When $\mathcal{D}$ correctly guess the bit b then $\mathcal{O}_2$ assumes its challenge oracle was returning real output from the hash function so it outputs $b' = 1$. Otherwise it outputs $b' = 0$. Thus, $\Pr[G_1] = \Pr[b' = 1 \,|\, b_{\mathsf{odhe}} = 1]$ and $\Pr[G_2] = \Pr[b' = 1 \,|\, b_{\mathsf{odhe}} = 0]$ from which it follows that

$\mathsf{Adv}^{\mathsf{odhe}}_{\mathbb{G},\mathsf{H},\mathcal{O}_2} = \Pr[G_1] - \Pr[G_2]$. As a result of the above and our previous sequence of inequalities, we get:

$$\begin{aligned}\Pr[\text{KIND}^{\mathcal{D}}_{\mathsf{RKE}}] &\leq q_{\text{RATSEND}} \cdot \mathsf{Adv}^{\mathsf{odhe}}_{\mathbb{G},\mathsf{H},\mathcal{O}_1} + (q_{\text{RATSEND}} + 1) \cdot \mathsf{Adv}^{\mathsf{sufcma}}_{\mathsf{F},\mathcal{F}} + \Pr[G_1] \\ &= q_{\text{RATSEND}} \cdot \mathsf{Adv}^{\mathsf{odhe}}_{\mathbb{G},\mathsf{H},\mathcal{O}_1} + (q_{\text{RATSEND}} + 1) \cdot \mathsf{Adv}^{\mathsf{sufcma}}_{\mathsf{F},\mathcal{F}} + \mathsf{Adv}^{\mathsf{odhe}}_{\mathbb{G},\mathsf{H},\mathcal{O}_2} + \Pr[G_2].\end{aligned}$$

Finally, $\Pr[G_2] = 1/2$ because the view of $\mathcal{D}$ is independent of b in G_2. To see this, first note that oracle CHSEND returns uniformly random bits regardless of the challenge bit. So we only need to verify that the CHREC returns the same random bits if its last if-statement is reached. This could only fail to occur if CHREC was called when restricted and unchanged are both false. However, flags restricted and unchanged can only be simultaneously false at the end of an oracle call to RATREC if they were already both false at the time when this oracle was called. Thus no call to RATREC can be the first to set them both to false.

This yields the claimed bound on the advantage of $\mathcal{D}$. The bounds on the number of

oracle queries made by the adversaries are obtained by examining their code. □

1.4 Ratcheted encryption

In this section we define ratcheted encryption schemes, and show how to construct them by composing ratcheted key exchange with symmetric encryption. This serves as a starting point for discussing ratcheted encryption, and we also discuss possible extensions.

Ratcheted encryption schemes.

Our definition of ratcheted encryption extends the definition of ratcheted key exchange by adding encryption and decryption algorithms. Ratcheted encryption schemes inherit the key generation algorithms from ratcheted key exchange schemes, and use the resulting shared keys as symmetric encryption keys. In line with our definition for ratcheted key exchange, we only consider one-sided ratcheted encryption, meaning that the sender uses its key only for encryption, and the receiver uses its key only for decryption.

A ratcheted encryption scheme RE specifies algorithms $\mathsf{RE.IKg}$, $\mathsf{RE.SKg}$, $\mathsf{RE.RKg}$, $\mathsf{RE.Enc}$ and $\mathsf{RE.Dec}$, where $\mathsf{RE.Enc}$ and $\mathsf{RE.Dec}$ are deterministic. Associated to RE is a nonce space $\mathsf{RE.NS}$, sender's key generation randomness space $\mathsf{RE.RS}$, and a ciphertext length function $\mathsf{RE.cl}\colon \mathbb{N} \to \mathbb{N}$. Initial key generation algorithm $\mathsf{RE.IKg}$ returns $k, sek_s, (stk_s, stk_r, sek_r)$, where k is an encryption key, stk_s, sek_s are a sender's static key and session key, and stk_r, sek_r are receiver's static key and receiver's session key, respectively. The sender's and receiver's (symmetric) encryption keys are initialized to $k_s = k_r = k$. Sender's key generation algorithm $\mathsf{RE.SKg}$ takes stk_s, sek_s and randomness $r \in \mathsf{RE.RS}$ to return a new sender's session key sek_s, a new sender's encryption key k_s, and update information upd. Receiver's key generation algorithm $\mathsf{RE.RKg}$ takes stk_s, stk_r, sek_r, upd and receiver's encryption key k_r to return a new receiver's session key sek_r, a new receiver's encryption key k_r, and a flag $acc \in \{\mathsf{true}, \mathsf{false}\}$. Encryption algorithm $\mathsf{RE.Enc}$ takes k_s, a nonce $n \in \mathsf{RE.NS}$, a plaintext message $m \in \{0,1\}^*$

and a header $h \in \{0,1\}^*$ to return a ciphertext $c \in \{0,1\}^{\mathsf{RE.cl}(|m|)}$. Decryption algorithm RE.Dec takes k_r, n, c, h to return $m \in \{0,1\}^* \cup \{\perp\}$.

Correctness of ratcheted encryption.

Correctness of ratcheted encryption extends that of ratcheted key exchange. It requires that messages encrypted using sender's key should correctly decrypt using the corresponding receiver's key.

Consider game RE-COR of Fig. 1.4 associated to a ratcheted encryption scheme R and an adversary $\mathcal{C}$, where $\mathcal{C}$ is provided with an access to oracles UP, RATREC and ENC. The advantage of $\mathcal{C}$ breaking the correctness of R is defined as $\mathsf{Adv}^{\mathsf{recor}}_{\mathsf{R},\mathcal{C}} = 1 - \Pr[\text{RE-COR}^{\mathcal{C}}_{\mathsf{R}}]$. Correctness property requires that $\mathsf{Adv}^{\mathsf{recor}}_{\mathsf{R},\mathcal{C}} = 0$ for all unbounded adversaries $\mathcal{C}$. Compared to the correctness game for ratcheted key exchange, the new element is that adversary $\mathcal{C}$ also gets access to an encryption oracle ENC, which can be queried to test the decryption correctness.

Ratcheted authenticated encryption.

Consider game RAE on the right side of Fig. 1.5 associated to a ratcheted encryption scheme RE and an adversary $\mathcal{A}$. It extends the security definition of ratcheted key exchange (as defined in game KIND on the left side of Fig. 1.5) by replacing oracles CHSEND and CHREC with oracles ENC and DEC. Oracles RATSEND, RATREC and EXP are the same in both games. Oracles ENC and DEC are defined as follows. In the real world (when $b = 1$) oracle ENC encrypts messages under the sender's key, and oracle DEC decrypts ciphertexts under the receiver's key. In the random world (when $b = 0$) oracle ENC returns uniformly random strings, and oracle DEC always returns an incorrect decryption symbol $\perp$. The goal of the adversary is to distinguish between the two cases. The advantage of $\mathcal{A}$ in breaking the RAE security of RE is defined as $\mathsf{Adv}^{\mathsf{rae}}_{\mathsf{RE},\mathcal{A}} = 2\Pr[\mathrm{RAE}^{\mathcal{A}}_{\mathsf{RE}}] - 1$.

We note that the adversary is only allowed to get a single encryption for each unique pair of (i_s, n). This restriction stems from the fact that most known nonce-based encryption schemes are not resistant to *nonce-misuse*. Our definition can be relaxed to only prevent queries where

Algorithm $\mathsf{RE.IKg}$

$(k, sek_s, (stk_s, stk_r, sek_r)) \leftarrow\$ \mathsf{RKE.IKg}$
Return $(k, sek_s, (stk_s, stk_r, sek_r))$

Algorithm $\mathsf{RE.SKg}(stk_s, sek_s; r)$

$(sek_s, k_s, upd) \leftarrow \mathsf{RKE.SKg}(stk_s, sek_s; r)$
Return (sek_s, k_s, upd)

Algorithm $\mathsf{RE.RKg}(stk_s, stk_r, sek_r, upd, k_r)$

$(sek_r, k_r, acc) \leftarrow\$ \mathsf{RKE.RKg}(stk_s, stk_r, sek_r, upd, k_r)$
Return (sek_r, k_r, acc)

Algorithm $\mathsf{RE.Enc}(k_s, n, m, h)$

$c \leftarrow \mathsf{SE.Enc}(k_s, n, m, h)$
Return c

Algorithm $\mathsf{RE.Dec}(k_r, n, c, h)$

$m \leftarrow \mathsf{SE.Dec}(k_r, n, c, h)$
Return m

Figure 1.13. Ratcheted encryption scheme $\mathsf{RE} = \mathsf{RATCHET\text{-}ENC}[\mathsf{RKE}, \mathsf{SE}]$.

(i_s, n, m) — or even (i_s, n, m, h) — are repeated, but it would increasingly limit the choice of the underlying symmetric schemes that can be used for this purpose (fewer schemes would satisfy stronger security definitions of multi-user authenticated encryption).

Revisiting the treatment of the restricted flag.

Similar to the definition of KIND, one could consider strengthening the definition of RAE by never resetting the restricted flag back to false (as discussed in Section 1.3.2). There would seem to be a more clear motivation to use the stronger definition in the case of encryption. Namely, our current security definition allows adversary to compromise the sender, use the exposed secrets to communicate with the receiver, and then restore the initial conversation link between the sender and the receiver. This represents an ability to stealthily insert arbitrary messages in the middle of someone's conversation, without ultimately disrupting the conversation. However, note that even a stronger definition (one that does not reset the restricted flag) appears to allow such attack, because the adversary might be able to compromise the sender and insert the messages before the next time the key ratcheting happens. The success of such attack would depend on how often the keys are being ratcheted.

Ratcheted encryption scheme RATCHET-ENC.

We build a ratcheted encryption scheme by combining a ratcheted key exchange scheme with a symmetric encryption scheme. In our composition the output keys of the ratcheted key

exchange scheme are used as encryption keys for the symmetric encryption scheme.

Let RKE be a ratcheted key exchange scheme and SE be a symmetric encryption scheme such that $\mathsf{SE.kl} = \mathsf{RKE.kl}$. From these two components we build a ratcheted encryption scheme $\mathsf{RE} = \mathsf{RATCHET\text{-}ENC}[\mathsf{RKE}, \mathsf{SE}]$ as defined in Fig. 1.13, with $\mathsf{RE.NS} = \mathsf{SE.NS}$, $\mathsf{RE.RS} = \mathsf{RKE.RS}$ and $\mathsf{RE.cl} = \mathsf{SE.cl}$.

Security of ratcheted encryption scheme $\mathsf{RATCHET\text{-}ENC}$.

The following says the security of encryption scheme $\mathsf{RE} = \mathsf{RATCHET\text{-}ENC}[\mathsf{RKE}, \mathsf{SE}]$ can be reduced to the KIND security of the ratcheted key exchange scheme RKE and MAE security of the symmetric encryption scheme SE.

Theorem 2. *Let RKE be a ratcheted key exchange scheme. Let SE be a symmetric encryption scheme such that $\mathsf{SE.kl} = \mathsf{RKE.kl}$. Let $\mathsf{RE} = \mathsf{RATCHET\text{-}ENC}[\mathsf{RKE}, \mathsf{SE}]$. Let $\mathcal{A}$ be an adversary attacking the* RAE*-security of* RE *that makes q_{RATSEND} queries to its* $\mathrm{RATSEND}$ *oracle, q_{RATREC} queries to its* RATREC *oracle, q_{EXP} queries to its* EXP *oracle, q_{ENC} queries to its* ENC *oracle, and q_{DEC} queries to its* DEC *oracle. Then there is an adversary $\mathcal{D}$ attacking the* KIND*-security of* RKE *and an adversary $\mathcal{N}$ attacking the* MAE*-security of* SE *such that*

$$\mathsf{Adv}^{\mathsf{rae}}_{\mathsf{RE},\mathcal{A}} \leq 2 \cdot \mathsf{Adv}^{\mathsf{kind}}_{\mathsf{RKE},\mathcal{D}} + \mathsf{Adv}^{\mathsf{mae}}_{\mathsf{SE},\mathcal{N}}.$$

Adversary $\mathcal{D}$ makes at most q_{EXP} queries to its EXP *oracle, q_{ENC} queries to its* CHSEND *oracle, q_{DEC} queries to its* CHREC *oracle, and the same number of queries as $\mathcal{A}$ to oracles* $\mathrm{RATSEND}$, RATREC. *Adversary $\mathcal{N}$ makes at most* $\max(q_{\mathrm{RATSEND}}, q_{\mathrm{RATREC}})$ *queries to its* NEW *oracle, q_{ENC} queries to its* ENC *oracle, and q_{DEC} queries to its* DEC *oracle. Each of $\mathcal{D}$, $\mathcal{N}$ has a running time approximately that of $\mathcal{A}$.*

The proof is given below. It proceeds in two steps, first using the KIND-security of RKE and then using the MAE-security of SE. Recall that our goal is to show that adversary $\mathcal{A}$ playing RAE security game against RE is unable to distinguish between the real world (when oracles ENC and DEC return real encryptions and decryptions) and the random world (when oracle ENC

returns random strings, and oracle DEC returns incorrect decryption symbol). In the first step of the proof, we use the KIND-security of ratcheted key exchange scheme RKE to switch from using real keys to random keys when calling oracles ENC and DEC in the real world of the RAE security game. We note that oracles $\text{RATSEND}, \text{RATREC}, \text{EXP}$ will still operate on the *real* keys after the first step, and the adversary $\mathcal{D}$ against KIND-security of RKE is able to simulate them using its own oracles. At this point, the keys used to answer queries to oracles ENC and DEC (in the game derived from the initial RAE security game) are random and independent of the keys used to answer queries to oracles $\text{RATSEND}, \text{RATREC}, \text{EXP}$. Thus for the second step of the proof, we can build an adversary $\mathcal{N}$ against the MAE-security of SE that will generate its own set of keys for ratcheted encryption scheme RE and use them to produce simulated answers for $\mathcal{A}$'s oracle queries to $\text{RATSEND}, \text{RATREC}, \text{EXP}$. Adversary $\mathcal{N}$ will answer $\mathcal{A}$'s oracle queries to ENC, DEC using the oracles provided by the MAE game, and relay $\mathcal{A}$'s output bit as the answer for its own security game.

Theorem 2. Consider games G_0, G_1 of Fig. 1.14. Lines not annotated with comments are common to both games. Game G_0 is equivalent to $\text{RAE}^{\mathcal{A}}_{\mathsf{RE}}$, so

$$\mathsf{Adv}^{\mathsf{rae}}_{\mathsf{RE},\mathcal{A}} = 2\Pr[\text{G}_0] - 1. \tag{1.2}$$

Game G_1 differs from game G_0 by using uniformly random keys to answer ENC and DEC oracle queries. Both games use real keys to answer EXP oracle queries.

First, we construct an adversary $\mathcal{D}$ against the KIND-security of RKE, as defined in Fig. 1.15. Adversary $\mathcal{D}$ simulates adversary $\mathcal{A}$ as follows. $\mathcal{A}$'s oracle queries to RATSEND, RATREC and EXP are directly answered by the corresponding $\mathcal{D}$'s oracles (but $\mathcal{D}$ also does some bookkeeping to maintain the states that are necessary for simulating other oracle queries). $\mathcal{D}$ simulates $\mathcal{A}$'s queries to ENC and DEC by calling it's own oracles CHSEND and CHREC and using the received challenge keys to encrypt and decrypt the messages itself. Let b denote the challenge bit in game $\text{KIND}^{\mathcal{D}}_{\mathsf{RKE}}$, and let b' denote the corresponding guess made by the adversary $\mathcal{D}$. We have $\Pr[\text{G}_0] = \Pr[b' = 1 \mid b = 1]$ and $\Pr[\text{G}_1] = \Pr[b' = 1 \mid b = 0]$. It follows that

Games G_0–G_1

$(k, sek_s, (stk_s, stk_r, sek_r)) \leftarrow\!\!\$\ \mathsf{RKE.IKg}$
$b \leftarrow\!\!\$\ \{0,1\}$; $i_s \leftarrow 0$; $i_r \leftarrow 0$; $k_s \leftarrow k$; $k_r \leftarrow k$
$b' \leftarrow\!\!\$\ \mathcal{A}^{\text{RatSend,RatRec,Exp,Enc,Dec}}(stk_s)$; Return $(b' = b)$

RatSend
$r \leftarrow\!\!\$\ \mathsf{RKE.RS}$; $(sek_s, k_s, upd) \leftarrow \mathsf{RKE.SKg}(stk_s, sek_s; r)$
$\mathsf{auth}[i_s] \leftarrow upd$; $i_s \leftarrow i_s + 1$; Return upd

RatRec(upd)
$(sek_r, k_r, acc) \leftarrow\!\!\$\ \mathsf{RKE.RKg}(stk_s, stk_r, sek_r, upd, k_r)$
If not acc then return false
If $\mathsf{op}[i_r] =$ "exp" then restricted $\leftarrow$ true
If $upd = \mathsf{auth}[i_r]$ then restricted $\leftarrow$ false
$i_r \leftarrow i_r + 1$; Return true

Exp
If $\mathsf{op}[i_s] =$ "ch" then return $\perp$
$\mathsf{op}[i_s] \leftarrow$ "exp" ; Return (r, sek_s, k_s)

Enc(n, m, h)
If $\mathsf{op}[i_s] =$ "exp" then return $\perp$
$\mathsf{op}[i_s] \leftarrow$ "ch"
If $(i_s, n) \in U$ then return $\perp$
$c_1 \leftarrow \mathsf{SE.Enc}(k_s, n, m, h)$ // G_0
If $\mathsf{rkey}[i_s] = \perp$ then $\mathsf{rkey}[i_s] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{RKE.kl}}$ // G_1
$c_1 \leftarrow \mathsf{SE.Enc}(\mathsf{rkey}[i_s], n, m, h)$ // G_1
$c_0 \leftarrow\!\!\$\ \{0,1\}^{\mathsf{RE.cl}(|m|)}$; $U \leftarrow U \cup \{(i_s, n)\}$
$S \leftarrow S \cup \{(i_s, n, c_b, h)\}$; Return c_b

Dec(n, c, h)
If restricted then return $\mathsf{SE.Dec}(k_r, n, c, h)$
If $\mathsf{op}[i_r] =$ "exp" then return $\perp$
$\mathsf{op}[i_r] \leftarrow$ "ch"
If $(i_r, n, c, h) \in S$ then return $\perp$
$m \leftarrow \mathsf{SE.Dec}(k_r, n, c, h)$ // G_0
If $\mathsf{rkey}[i_r] = \perp$ then $\mathsf{rkey}[i_r] \leftarrow\!\!\$\ \{0,1\}^{\mathsf{RKE.kl}}$ // G_1
$m \leftarrow \mathsf{SE.Dec}(\mathsf{rkey}[i_r], n, c, h)$ // G_1
If $b = 1$ then return m else return $\perp$

Figure 1.14. Games $\mathrm{G}_0, \mathrm{G}_1$ for proof of Theorem 2.

$$\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_1] = \mathsf{Adv}^{\mathsf{kind}}_{\mathsf{RKE},\mathcal{D}}. \tag{1.3}$$

Next, we construct an adversary $\mathcal{N}$ against the MAE-security of SE, as defined in

Adversary $\mathcal{D}^{\text{RatSend},\text{RatRec},\text{Exp},\text{ChSend},\text{ChRec}}(stk_s)$

$b \leftarrow\$ \{0,1\}$; $i_s \leftarrow 0$; $i_r \leftarrow 0$
$b' \leftarrow\$ \mathcal{A}^{\text{RatSendSim},\text{RatRecSim},\text{ExpSim},\text{EncSim},\text{DecSim}}(stk_s)$
If $(b' = b)$ then return 1 else return 0

$\text{EncSim}(n,m,h)$

If $\mathsf{op}[i_s] =$ "exp" then return $\perp$
$\mathsf{op}[i_s] \leftarrow$ "ch"
If $(i_s,n) \in U$ then return $\perp$
$k_s \leftarrow \text{ChSend}$; $c_1 \leftarrow \mathsf{SE.Enc}(k_s,n,m,h)$
$c_0 \leftarrow\$ \{0,1\}^{\mathsf{RE.cl}(|m|)}$; $U \leftarrow U \cup \{(i_s,n)\}$
$S \leftarrow S \cup \{(i_s,n,c_b,h)\}$; Return c_b

$\text{DecSim}(n,c,h)$

If restricted then
  $k_r \leftarrow \text{ChRec}$; Return $\mathsf{SE.Dec}(k_r,n,c,h)$
If $\mathsf{op}[i_r] =$ "exp" then return $\perp$
$\mathsf{op}[i_r] \leftarrow$ "ch"
If $(i_r,n,c,h) \in S$ then return $\perp$
$k_r \leftarrow \text{ChRec}$; $m \leftarrow \mathsf{SE.Dec}(k_r,n,c,h)$
If $b = 1$ then return m else return $\perp$

RatSendSim

$upd \leftarrow \text{RatSend}$
$\mathsf{auth}[i_s] \leftarrow upd$
$i_s \leftarrow i_s + 1$
Return upd

$\text{RatRecSim}(upd)$

success $\leftarrow \text{RatRec}(upd)$
If success then
  $u_0 \leftarrow (\mathsf{op}[i_r] =$ "exp"$)$
  $u_1 \leftarrow (upd = \mathsf{auth}[i_r])$
  If u_0 then restricted $\leftarrow$ true
  If u_1 then restricted $\leftarrow$ false
  $i_r \leftarrow i_r + 1$
Return success

ExpSim

If $\mathsf{op}[i_s] =$ "ch" then return $\perp$
$\mathsf{op}[i_s] \leftarrow$ "exp"
Return Exp

Figure 1.15. Adversary $\mathcal{D}$ for proof of Theorem 2.

Fig. 1.16. Adversary $\mathcal{N}$ generates its own keys for the ratcheted key exchange scheme RKE, and uses them to answer $\mathcal{A}$'s queries to oracles RatSend, RatRec and Exp (as well as $\mathcal{A}$'s queries to Dec in the case when restricted is true). Furthermore, $\mathcal{A}$'s calls to Enc and Dec are answered using the corresponding oracles that are provided to $\mathcal{N}$ in game MAE. We have $\Pr[G_1] = \text{MAE}_{\mathsf{SE}}^{\mathcal{N}}$, so

$$\mathsf{Adv}_{\mathsf{SE},\mathcal{N}}^{\mathsf{mae}} = 2\Pr[G_1] - 1. \tag{1.4}$$

The theorem statement follows from equations (1.2)–(1.4). □

Extensions.

We defined our encryption schemes to be one-sided in both communication (meaning that the messages are assumed to be sent only in one direction, from the sender to the receiver), and in security (only protecting against the exposure of the sender's secrets). It would be useful to consider *two-sided* communication (but still one-sided security). In our model the sender and

Adversary $\mathcal{N}^{\text{NEW,ENC,DEC}}$

$(k, sek_s, (stk_s, stk_r, sek_r)) \leftarrow\!\$\ \mathsf{RKE.IKg}$
$v \leftarrow 0;\ i_s \leftarrow 0;\ i_r \leftarrow 0;\ k_s \leftarrow k;\ k_r \leftarrow k$
$b' \leftarrow\!\$\ \mathcal{A}^{\text{RATSENDSIM,RATRECSIM,EXPSIM,ENCSIM,DECSIM}}(stk_s)$
Return b'

RATSENDSIM

$r \leftarrow\!\$\ \mathsf{RKE.RS};\ z \leftarrow \mathsf{RKE.SKg}(stk_s, sek_s; r)$
$(sek_s, k_s, upd) \leftarrow z;\ \mathsf{auth}[i_s] \leftarrow upd;\ i_s \leftarrow i_s + 1$
While $v < i_s$ do
  NEW; $v \leftarrow v + 1$
Return upd

RATRECSIM(upd)

$(sek_r, k_r, acc) \leftarrow\!\$\ \mathsf{RKE.RKg}(stk_s, stk_r, sek_r, upd, k_r)$
If not acc then return false
If $\mathsf{op}[i_r] =$ "exp" then restricted $\leftarrow$ true
If $upd = \mathsf{auth}[i_r]$ then restricted $\leftarrow$ false
$i_r \leftarrow i_r + 1$
While $v < i_r$ do
  NEW; $v \leftarrow v + 1$
Return true

EXPSIM

If $\mathsf{op}[i_s] =$ "ch" then return $\perp$
$\mathsf{op}[i_s] \leftarrow$ "exp"
Return (r, sek_s, k_s)

ENC(n, m, h)

If $\mathsf{op}[i_s] =$ "exp" then return $\perp$
$\mathsf{op}[i_s] \leftarrow$ "ch"
Return ENC(i_s, n, m, h)

DEC(n, c, h)

If restricted then
  Return $\mathsf{SE.Dec}(k_r, n, c, h)$
If $\mathsf{op}[i_r] =$ "exp" then return $\perp$
$\mathsf{op}[i_r] \leftarrow$ "ch"
Return DEC(i_r, n, c, h)

Figure 1.16. Adversary $\mathcal{N}$ for proof of Theorem 2.

---

the receiver already share the same key, but one would need to update the security game to allow using either key for encryption and decryption.

An important goal in studying ratcheted encryption is to model the *Double Ratchet algorithm* [48, 36] used in multiple real-world messaging applications, such as in WhatsApp [66] and in the Secret Conversations mode of Facebook Messenger [37]. This work models the asymmetric layer of key ratcheting, whereas the real-world applications also have a second layer of key ratcheting that happens in a symmetric setting. In our model, this can be possibly achieved by using the output keys of ratcheted key exchange to initialize a *forward-secure* symmetric encryption scheme. We do not capture this possibility; both the syntax and the security definitions would need to be significantly extended.

1.5 Oracle Diffie-Hellman with Exposures in ROM

In this section we justify the plausibility of the Oracle Diffie-Hellman with Exposures (ODHE) assumption that was introduced in Section 1.2. We used ODHE to prove the security of our ratcheted key exchange scheme in Section 1.3.4. We now show that the ODHE assumption reduces to the Strong Computational Diffie-Hellman (SCDH) assumption when the hash function in the former is modeled as the random oracle.

We do the reduction in two steps. We introduce the *Strong Computational Diffie-Hellman with Exposures* (SCDHE) assumption and use it as an intermediate assumption. In the first step, we show that ODHE reduces to SCDHE when the hash function in ODHE is modeled as the random oracle. In the second step, we show that SCDHE reduces to SCDH. We provide two different reductions for the second step. Our first reduction uses a standard index guessing proof and thus creates a factor q loss in the advantage. Our second reduction avoids this multiplicative advantage loss by defining a "rewinding" adversary that uses the self-reducibility of Diffie-Hellman problems.

We now define the necessary assumptions and state two alternative theorems as outlined above.

Random Oracle Model.

In the first step of our reduction we will work in the random oracle model (ROM) [14], modeling a hash function as the random oracle. The random oracle RO models a truly random function and is defined as follows:

$$\underline{\mathrm{RO}(z,\kappa)}$$

$$\text{If } T[z,\kappa] = \bot \text{ then } T[z,\kappa] \leftarrow\!\!\$\ \{0,1\}^{\kappa}$$

$$\text{Return } T[z,\kappa]$$

It takes a string $z \in \{0,1\}^*$ and an output length $\kappa \in \mathbb{N}$ as input, to return an element from $\{0,1\}^{\kappa}$. We prove our claims for a hash function that simply evaluates the random oracle on its inputs. We now extend the ODHE assumption from Section 1.2 to the random oracle model.

Game $\mathrm{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H}}$

$b \leftarrow\$ \{0,1\}$; $hk \leftarrow\$ \{0,1\}^{\mathsf{H.kl}}$; $g \leftarrow\$ \mathbb{G}^*$; $y \leftarrow\$ \mathbb{Z}_p$; $v \leftarrow -1$
$b' \leftarrow\$ \mathcal{O}^{\textsc{Up},\textsc{Ch},\textsc{Exp},\textsc{Hash},\mathrm{RO}}(hk, g, g^y)$; Return $(b' = b)$

$\underline{\textsc{Up}}$

$\mathsf{op} \leftarrow \varepsilon$; $v \leftarrow v+1$; $x[v] \leftarrow\$ \mathbb{Z}_p$; Return $g^{x[v]}$

$\underline{\textsc{Ch}(s)}$

If $(\mathsf{op} = \text{“exp”})$ or $((v, s, g^{x[v]}) \in S_{\mathsf{hash}})$ then return $\perp$
$\mathsf{op} \leftarrow \text{“ch”}$; $S_{\mathsf{ch}} \leftarrow S_{\mathsf{ch}} \cup \{(v, s, g^{x[v]})\}$; $e \leftarrow g^{x[v]\cdot y}$
If $\mathsf{mem}[v, s, e] = \perp$ then $\mathsf{mem}[v, s, e] \leftarrow\$ \{0,1\}^{\mathsf{H.ol}}$
$r_1 \leftarrow \mathsf{H.Ev}^{\mathrm{RO}}(hk, v \,\|\, s \,\|\, e)$; $r_0 \leftarrow \mathsf{mem}[v, s, e]$; Return r_b

$\underline{\textsc{Exp}}$

If $\mathsf{op} = \text{“ch”}$ then return $\perp$
$\mathsf{op} \leftarrow \text{“exp”}$; Return $x[v]$

$\underline{\textsc{Hash}(i, s, X)}$

If $(i, s, X) \in S_{\mathsf{ch}}$ then return $\perp$
If $i = v$ then $S_{\mathsf{hash}} \leftarrow S_{\mathsf{hash}} \cup \{(i, s, X)\}$
Return $\mathsf{H.Ev}^{\mathrm{RO}}(hk, i \,\|\, s \,\|\, X^y)$

$\underline{\mathrm{RO}(z, \kappa)}$

If $T[z, \kappa] = \perp$ then $T[z, \kappa] \leftarrow\$ \{0,1\}^{\kappa}$
Return $T[z, \kappa]$

Figure 1.17. Game defining Oracle Diffie-Hellman with Exposures in ROM assumption for $\mathbb{G}, \mathsf{H}$.

Oracle Diffie-Hellman with Exposures in ROM assumption.

Let $\mathbb{G}$ be a cyclic group of order $p \in \mathbb{N}$, and let $\mathbb{G}^*$ denote the set of its generators. Let H be a function family such that $\mathsf{H.In} = \{0,1\}^*$. Consider game ODHER of Fig. 1.17 associated to $\mathbb{G}, \mathsf{H}$ and an adversary $\mathcal{O}$, where $\mathcal{O}$ is required to call oracle $\textsc{Up}$ at least once prior to making any oracle queries to $\textsc{Ch}$ and $\textsc{Exp}$. This game is similar to the game ODHE from Section 1.2, except that it provides adversary $\mathcal{O}$ and hash function H with an access to the random oracle RO. The advantage of $\mathcal{O}$ in breaking the ODHER security of $\mathbb{G}, \mathsf{H}$ is defined as $\mathsf{Adv}^{\mathsf{odher}}_{\mathbb{G},\mathsf{H},\mathcal{O}} = 2\Pr[\mathrm{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H}}] - 1$.

Game $\text{SCDH}_{\mathbb{G}}^{\mathcal{S}}$	Game $\text{SCDHE}_{\mathbb{G}}^{\mathcal{B}}$
$g \leftarrow\$ \mathbb{G}^*$; $y \leftarrow\$ \mathbb{Z}_p$; $x \leftarrow\$ \mathbb{Z}_p$	$g \leftarrow\$ \mathbb{G}^*$; $y \leftarrow\$ \mathbb{Z}_p$; $v \leftarrow -1$
$Z \leftarrow\$ \mathcal{S}^{\text{DH}}(g, g^x, g^y)$	$(j, Z) \leftarrow\$ \mathcal{B}^{\text{DH,UP,EXP}}(g, g^y)$
Return $(Z = g^{xy})$	valid $\leftarrow (0 \leq j \leq v)$ and $(\mathsf{op}[j] \neq \text{“exp”})$
	Return valid and $(Z = g^{x[j] \cdot y})$
$\underline{\text{DH}(X, Z)}$	$\underline{\text{DH}(X, Z)}$
Return $(X^y = Z)$	Return $(X^y = Z)$
	$\underline{\text{UP}}$
	$v \leftarrow v + 1$; $x[v] \leftarrow\$ \mathbb{Z}_p$; Return $g^{x[v]}$
	$\underline{\text{EXP}}$
	$\mathsf{op}[v] \leftarrow$ “exp” ; Return $x[v]$

Figure 1.18. Games defining Strong Computational Diffie-Hellman assumption in group $\mathbb{G}$, and Strong Computational Diffie-Hellman with Exposures assumption in group $\mathbb{G}$.

Strong Computational Diffie-Hellman assumption.

Let $\mathbb{G}$ be a cyclic group of order $p \in \mathbb{N}$, and let $\mathbb{G}^*$ denote the set of its generators. Consider game SCDH of Fig. 1.18, associated to group $\mathbb{G}$ and to an adversary $\mathcal{S}$. Adversary $\mathcal{S}$ receives g, g^x, g^y as input, where g is a group generator and g^x, g^y are random group elements for some secret values x, y. It is also provided with an oracle DH that takes arbitrary group elements X, Z and returns whether $X^y = Z$. The adversary wins the game if it can compute the value of g^{xy}. The advantage of $\mathcal{S}$ in breaking the SCDH security of $\mathbb{G}$ is defined as $\mathsf{Adv}^{\mathsf{scdh}}_{\mathbb{G},\mathcal{S}} = \Pr[\text{SCDH}^{\mathcal{S}}_{\mathbb{G}}]$. The SCDH assumption was originally defined in [1].

Strong Computational Diffie-Hellman with Exposures assumption.

Let $\mathbb{G}$ be a cyclic group of order $p \in \mathbb{N}$, and let $\mathbb{G}^*$ denote the set of its generators. Consider game SCDHE of Fig. 1.18, associated to group $\mathbb{G}$ and to an adversary $\mathcal{B}$. Adversary $\mathcal{B}$ receives g, g^y as input, where g is a group generator and g^y is a random group element for some secret value y. It is also provided with oracles DH, UP, EXP defined as follows. The DH oracle is identical to the one from the SCDH game; it takes arbitrary group elements X, Z and returns whether $X^y = Z$. The update oracle UP generates a new random group exponent $x[v]$ and returns $g^{x[v]}$, where v is a counter that enumerates all challenge exponents (indexed from 0). The expose

oracle $\text{E}\textsc{xp}$ returns $x[v]$ (we do not require that $\mathcal{B}$ calls $\text{U}\textsc{p}$ prior to its first call to $\text{E}\textsc{xp}$, meaning that $\mathcal{B}$ is allowed to expose the unitialized value at the location -1 of map x). The adversary wins the game if it returns a pair (j, Z) such that $Z = g^{x[j]\cdot y}$ and $x[j]$ was not exposed. The advantage of $\mathcal{B}$ in breaking the SCDHE security of $\mathbb{G}$ is defined as $\mathsf{Adv}^{\mathsf{scdhe}}_{\mathbb{G},\mathcal{B}} = \Pr[\text{SCDHE}^{\mathcal{B}}_{\mathbb{G}}]$.

Theorem statements.

We prove the following theorems. They relate the advantage of an adversary $\mathcal{O}$ against the ODHER security of $\mathbb{G}, \mathsf{H}$ —where H is instantiated by the random oracle— to the advantage of an adversary $\mathcal{S}$ against the SCDH security in $\mathbb{G}$. The second theorem achieves a better upper-bound for the advantage of $\mathcal{O}$, but its proof is more involved and the constructed adversary $\mathcal{S}$ has a worse running time.

We prove our claims for a function family H which is defined by $\mathsf{H.Ev}^{\mathrm{RO}}(hk, s) = \mathrm{RO}(hk \,\|\, s, \mathsf{H.ol})$ for some $\mathsf{H.ol} \in \mathbb{N}$. Note that the evaluation algorithm of a function family is defined to be deterministic, whereas RO is a randomized procedure. For H to be well-defined, one has to be careful when defining deterministic algorithms. For our purposes, a deterministic algorithms is one that takes no random coins.

Theorem 3. *Let $\mathbb{G}$ be a cyclic group of order $p \in \mathbb{N}$, and let $\mathbb{G}^*$ denote the set of its generators. Let H be a function family defined by $\mathsf{H.Ev}^{\mathrm{RO}}(hk, s) = \mathrm{RO}(hk \,\|\, s, \mathsf{H.ol})$ for some $\mathsf{H.ol} \in \mathbb{N}$. Let $\mathcal{O}$ be an adversary attacking the ODHER-security of $\mathbb{G}, \mathsf{H}$ that makes $q_{\text{U}\textsc{p}}$ queries to its $\text{U}\textsc{p}$ oracle, $q_{\text{C}\textsc{h}}$ queries to its $\text{C}\textsc{h}$ oracle, $q_{\text{H}\textsc{ash}}$ queries to its $\text{H}\textsc{ash}$ oracle, and q_{RO} queries to its RO oracle. Then there is an adversary $\mathcal{S}$ attacking the SCDH-security of $\mathbb{G}$ such that*

$$\mathsf{Adv}^{\mathsf{odher}}_{\mathbb{G},\mathsf{H},\mathcal{O}} \leq q_{\text{U}\textsc{p}} \cdot \mathsf{Adv}^{\mathsf{scdh}}_{\mathbb{G},\mathcal{S}} + q_{\text{H}\textsc{ash}}/p.$$

Adversary $\mathcal{S}$ makes at most $q_{\mathrm{RO}} \cdot (q_{\text{C}\textsc{h}} + q_{\text{H}\textsc{ash}})$ queries to its DH oracle. Its running time is approximately that of $\mathcal{O}$ plus extra terms that are dominated by $q_{\mathrm{RO}} \cdot (q_{\text{C}\textsc{h}} + q_{\text{H}\textsc{ash}})$.

To prove this theorem, Section 1.5.1 provides the following. First, it reduces ODHE to SCDHE where the hash function in the former is modeled as the random oracle. Second, it

reduces SCDHE to SCDH in the standard model.

Theorem 4. *Let $\mathbb{G}$ be a cyclic group of order $p \in \mathbb{N}$, and let $\mathbb{G}^*$ denote the set of its generators. Let H be a function family defined by $\mathsf{H.Ev}^{\mathrm{RO}}(hk,s) = \mathrm{RO}(hk \| s, \mathsf{H.ol})$ for some $\mathsf{H.ol} \in \mathbb{N}$. Let $\mathcal{O}$ be an adversary attacking the* ODHER*-security of* $\mathbb{G}, \mathsf{H}$ *that makes q_{UP} queries to its* UP *oracle, q_{CH} queries to its* CH *oracle, q_{EXP} queries to its* EXP *oracle, q_{HASH} queries to its* HASH *oracle, and q_{RO} queries to its* RO *oracle. Let $u \in \mathbb{N}$. Then there is an adversary $\mathcal{S}_u$ attacking the* SCDH*-security of* $\mathbb{G}$ *such that*

$$\mathsf{Adv}^{\mathsf{odher}}_{\mathbb{G},\mathsf{H},\mathcal{O}} \leq \mathsf{Adv}^{\mathsf{scdh}}_{\mathbb{G},\mathcal{S}_u} + q_{\mathrm{UP}} \cdot 2^{-u} + q_{\mathrm{HASH}}/p.$$

Let $q_0 = q_{\mathrm{RO}} \cdot (q_{\mathrm{CH}} + q_{\mathrm{HASH}})$ and $q_1 = 1 + u \cdot q_{\mathrm{UP}}$. Adversary $\mathcal{S}_u$ makes at most $q_0 \cdot q_1$ queries to its DH *oracle. Its running time is approximately q_1 times that of $\mathcal{O}$ plus extra terms that are dominated by $q_0 \cdot q_1$.*

In Section 1.5.2 we provide an alternative reduction from SCDHE to SCDH that is more involved than the one in Section 1.5.1. The claims in Theorem 4 are implied by the random oracle reduction from ODHE to SCDHE in Section 1.5.1, along with the improved reduction from SCDHE to SCDH in Section 1.5.2.

1.5.1 ODHE reduction to SCDH in ROM

We now state and prove two lemmas that together imply the claim in Theorem 3. The first lemma reduces from ODHE to SCDHE in ROM, by showing how to use any ODHER adversary $\mathcal{O}$ to construct an SCDHE adversary $\mathcal{B}$. Then the second lemma reduces from any SCDHE adversary $\mathcal{B}$ to an adversary $\mathcal{S}$ against SCDH.

Reducing ODHE to SCDHE in ROM.

The first intermediate lemma is as follows.

Lemma 5. *Let $\mathbb{G}$ be a cyclic group of order $p \in \mathbb{N}$ and let $\mathbb{G}^*$ denote the set of its generators. Let H be a function family defined by $\mathsf{H.Ev}^{\mathrm{RO}}(hk,s) = \mathrm{RO}(hk \| s, \mathsf{H.ol})$ for some $\mathsf{H.ol} \in \mathbb{N}$. Let $\mathcal{O}$*

be an adversary attacking the ODHER*-security of* $\mathbb{G}, \mathsf{H}$ *that makes* q_{UP} *queries to its* UP *oracle,* q_{CH} *queries to its* CH *oracle,* q_{EXP} *queries to its* EXP *oracle,* q_{HASH} *queries to its* HASH *oracle, and* q_{RO} *queries to its* RO *oracle. Then there is an adversary* $\mathcal{B}$ *attacking the* SCDHE*-security of* $\mathbb{G}$ *such that*

$$\mathsf{Adv}^{\mathsf{odher}}_{\mathbb{G},\mathsf{H},\mathcal{O}} \leq \mathsf{Adv}^{\mathsf{scdhe}}_{\mathbb{G},\mathcal{B}} + q_{\text{HASH}}/p.$$

Adversary $\mathcal{B}$ *makes at most* q_{UP} *queries to its* UP *oracle,* q_{EXP} *queries to its* EXP *oracle, and* $q_{\text{RO}} \cdot (q_{\text{CH}} + q_{\text{HASH}})$ *queries to its* DH *oracle. Its running time is approximately that of* $\mathcal{O}$ *plus extra terms that are dominated by* $q_{\text{RO}} \cdot (q_{\text{CH}} + q_{\text{HASH}})$*.*

This lemma captures the intuition that the only way for $\mathcal{O}$ to distinguish the output of the hash function from random is if it was able to calculate $g^{x[v]\cdot y}$ and query this value to its random oracle. The q_{HASH}/p-term (erroneously omitted in earlier versions of this paper) bounds the unlikely event that $\mathcal{O}$ makes a query of the form $\text{HASH}(i, s, g^{x[i]\cdot y})$ *before* $x[i]$ was picked by the game.

Lemma 5. Let $\text{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H},b}$ denote the game which proceeds as $\text{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H}}$ with the challenge bit hard-coded to b and then returns true iff $b' = 1$, where b' is the bit returned by $\mathcal{O}$. A standard conditioning argument gives that

$$\mathsf{Adv}^{\mathsf{odher}}_{\mathbb{G},\mathsf{H},\mathcal{O}} = \Pr[\text{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H},1}] - \Pr[\text{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H},0}]. \tag{1.5}$$

Consider games G_0, G_1, and G_2 in Fig. 1.19. Lines annotated with game names in comments are only in the indicated games; other lines are common to all games. The result will follow by establishing the following claims.

(1) $\Pr[\text{G}_0] = \Pr[\text{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H},1}]$

(2) $\Pr[\text{G}_0] - \Pr[\text{G}_1] \leq q_{\text{HASH}}/p$

(3) $\Pr[\text{G}_1] - \Pr[\text{G}_2] \leq \mathsf{Adv}^{\mathsf{scdhe}}_{\mathbb{G},\mathcal{B}}$ (for $\mathcal{B}$ defined below)

(4) $\Pr[\text{G}_2] = \Pr[\text{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H},0}]$

$\underline{\text{Games } G_0, G_1, G_2}$

$hk \leftarrow\$ \{0,1\}^{\mathsf{H.kl}}$; $g \leftarrow\$ \mathbb{G}^*$; $y \leftarrow\$ \mathbb{Z}_p$; $v \leftarrow -1$
$b' \leftarrow\$ \mathcal{O}^{\text{UP,CH,EXP,HASH,RO}}(hk, g, g^y)$
Return $(b' = 1)$

$\underline{\text{UP}}$

$\mathsf{op} \leftarrow \varepsilon$; $v \leftarrow v+1$; $x[v] \leftarrow\$ \mathbb{Z}_p$; Return $g^{x[v]}$

$\underline{\text{CH}(s)}$

If $(\mathsf{op} = \text{“exp”})$ or $((v, s, g^{x[v]}) \in S_{\mathsf{hash}})$ then
  Return $\perp$
$\mathsf{op} \leftarrow \text{“ch”}$; $S_{\mathsf{ch}} \leftarrow S_{\mathsf{ch}} \cup \{(v, s, g^{x[v]})\}$; $e \leftarrow g^{x[v]\cdot y}$
If $T_{\text{CH}}[v,s,e] = \perp$ then
  $T_{\text{CH}}[v,s,e] \leftarrow\$ \{0,1\}^{\mathsf{H.ol}}$
  If $T_H[v,s,e] \neq \perp$ then //$(v,s,e) \in S'_{\mathsf{hash}}$
    $\mathsf{bad}_{0,1} \leftarrow \mathsf{true}$
    $T_{\text{CH}}[v,s,e] \leftarrow T_H[v,s,e]$ // G_0
  If $T[hk \| v \| s \| e, \mathsf{H.ol}] \neq \perp$ then
    $\mathsf{bad}_{1,2} \leftarrow \mathsf{true}$
    $T_{\text{CH}}[v,s,e] \leftarrow T[hk \| v \| s \| e, \mathsf{H.ol}]$ // G_0, G_1
$r \leftarrow T_{\text{CH}}[v,s,e]$
Return r

$\underline{\text{EXP}}$

If $\mathsf{op} = \text{“ch”}$ then return $\perp$
$\mathsf{op} \leftarrow \text{“exp”}$; Return $x[v]$

$\underline{\text{HASH}(i,s,X)}$

If $(i,s,X) \in S_{\mathsf{ch}}$ then return $\perp$
If $i < v$ then $S'_{\mathsf{hash}} \leftarrow S'_{\mathsf{hash}} \cup \{(i,s,X)\}$
If $i = v$ then $S_{\mathsf{hash}} \leftarrow S_{\mathsf{hash}} \cup \{(i,s,X)\}$
If $T_H[i,s,X^y] = \perp$ then
  $T_H[i,s,X^y] \leftarrow\$ \{0,1\}^{\mathsf{H.ol}}$
  If $T_{\text{CH}}[i,s,X^y] \neq \perp$ then
    $\mathsf{bad}_{0,1} \leftarrow \mathsf{true}$
    $T_H[i,s,X^y] \leftarrow T_{\text{CH}}[i,s,X^y]$ // G_0
  If $T[hk \| i \| s \| X^y, \mathsf{H.ol}] \neq \perp$ then
    $T_H[i,s,X^y] \leftarrow T[hk \| i \| s \| X^y, \mathsf{H.ol}]$
Return $T_H[i,s,X^y]$

$\underline{\text{RO}(z,\kappa)}$

If $T[z,\kappa] = \perp$ then
  $T[z,\kappa] \leftarrow\$ \{0,1\}^{\kappa}$
  $hk' \| i \| s \| e \leftarrow z$
  If $(hk', \kappa) = (hk, \mathsf{H.ol})$ then
    If $T_{\text{CH}}[i,s,e] \neq \perp$ then
      $\mathsf{bad}_{1,2} \leftarrow \mathsf{true}$
      $T[z,\kappa] \leftarrow T_{\text{CH}}[i,s,e]$ // G_0, G_1
    If $T_H[i,s,e] \neq \perp$ then
      $T[z,\kappa] \leftarrow T_H[i,s,e]$
Return $T[z,\kappa]$

Figure 1.19. Games G_0, G_1 for proof of Lemma 5.

The relevant calculation is as follows.

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{odher}}_{\mathbb{G},\mathsf{H},\mathcal{O}} &= \Pr[\text{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H},1}] - \Pr[\text{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H},0}] \\
&= \Pr[G_0] - \Pr[G_2] \\
&= (\Pr[G_0] - \Pr[G_1]) + (\Pr[G_1] - \Pr[G_2]) \\
&\leq q_{\text{HASH}}/p + \mathsf{Adv}^{\mathsf{scdhe}}_{\mathbb{G},\mathcal{B}}
\end{aligned}$$

Claim (1). We describe the process by which $\text{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H},1}$ can be modified to obtain G_0, establishing their first claim. First, all uses of mem are erased because they are unused and the

variable r_1 is renamed to r. Additionally, for expositional purposes we introduced a set S'_{hash} which is used on the second line of HASH to track queries made with $i < v$.

The more interesting changes came in our modification of how the random oracle is treated. In $\mathrm{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H},1}$ there are three ways the random oracle can be queried. The adversary can query it directly via RO and the game will query it during executions of H.Ev in CH and HASH. In G_0 separate random oracle tables (T, T_{CH}, and T_H respectively) are kept for for each of these query types. To ensure this does not create inconsistency in the random oracle each time one of these tables would be used we first check if a matching entry has been created in another one of these tables . The value in the first table sampled for some entry gets propagated to the corresponding entries of the other tables when needed. We have used highlighting to indicate the start of the table management in each of these oracles. These changes were designed to leave the game's behavior unchanged so $\Pr[\mathrm{G}_0] = \Pr[\mathrm{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H},1}]$ holds.

Claim (2). In transitioning from G_0 to G_2 we will be making T_H and T independent from T_{CH} which results in a game identical to $\mathrm{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H},0}$. Towards that, the difference between G_0 and G_1 is that in G_1 consistency between T_{CH} and T_H is no longer maintained. They are identical until the flag $\mathsf{bad}_{0,1}$ is set true. By the fundamental lemma of game playing [16], we have

$$\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_1] \leq \Pr[\mathsf{bad}^{\mathrm{G}_1}_{0,1}],$$

where $\Pr[\mathsf{bad}^{\mathrm{G}_1}_{0,1}]$ denotes the probability of setting $\mathsf{bad}_{0,1}$ true in game G_0.

Consider the possibility of $\mathsf{bad}_{0,1}$ being set in HASH. For this to occur it must hold that $(i,s,X) \notin S_{\mathsf{ch}}$ and $T_{\mathrm{CH}}[i,s,X^y] \neq \bot$ because of the earlier if statements. However, $T_{\mathrm{CH}}[i,s,X^y]$ can only be set *after* (i,s,X) was added to S_{ch} in CH. Thus $\mathsf{bad}_{0,1}$ being set in HASH is not possible.

Now consider the possibility of $\mathsf{bad}_{0,1}$ being set in CH. For this to occur it must hold that $(v,s,g^{x[v]}) \notin S_{\mathsf{hash}}$ and $T_H[v,s,g^{x[v]\cdot y}] \neq \bot$. The latter tells us that a query of $(v,s,g^{x[v]\cdot y})$ must have previously made to HASH, while the former tells us that the query must have been made

Adversary $\mathcal{B}^{\mathrm{DH},\textsc{Up},\textsc{Exp}}(g,Y)$

$hk \leftarrow_\$ \{0,1\}^{\mathsf{H.kl}}$; $v \leftarrow -1$;
$b' \leftarrow_\$ \mathcal{O}^{\textsc{UpSim},\textsc{ChSim},\textsc{ExpSim},\textsc{HashSim},\mathrm{ROSim}}(hk,g,g^y)$
Return $(\bot,\bot)$

$\textsc{UpSim}$

$\mathsf{op} \leftarrow \varepsilon$; $v \leftarrow v+1$; $X_{cur} \leftarrow \textsc{Up}$;

$\textsc{ChSim}(s)$

If $(\mathsf{op} = \text{“exp”})$ or $((v,s,X_{cur}) \in S_{\mathsf{hash}})$ then
  Return $\bot$
$\mathsf{op} \leftarrow \text{“ch”}$; $S_{\mathsf{ch}} \leftarrow S_{\mathsf{ch}} \cup \{(v,s,X_{cur})\}$
If $T_{\mathrm{CH}}[v,s,X_{cur}] = \bot$ then
    $T_{\mathrm{CH}}[v,s,X_{cur}] \leftarrow_\$ \{0,1\}^{\mathsf{H.ol}}$
  For each $(i',s',e) \in S_{RO}$ do
    If $(i',s') = (v,s)$ and $\mathrm{DH}(X_{cur},e)$ then
      abort(v,e)
$r \leftarrow T_{\mathrm{CH}}[v,s,X_{cur}]$
Return r

$\textsc{ExpSim}$

If $\mathsf{op} = \text{“ch”}$ then return $\bot$
$\mathsf{op} \leftarrow \text{“exp”}$; $x \leftarrow \textsc{Exp}$; Return x

$\textsc{HashSim}(i,s,X)$

If $(i,s,X) \in S_{\mathsf{ch}}$ then return $\bot$
If $i = v$ then $S_{\mathsf{hash}} \leftarrow S_{\mathsf{hash}} \cup \{(i,s,X)\}$
$S_H \leftarrow S_H \cup \{(i,s,X)\}$
If $T_H[i,s,X] = \bot$
  $T_H[i,s,X] \leftarrow_\$ \{0,1\}^{\mathsf{H.ol}}$
  For each $(i',s',e) \in S_{RO}$ do
    If $(i',s') = (i,s)$ and $\mathrm{DH}(X,e)$ then
      $T_H[i,s,X] \leftarrow T[hk \,\|\, i \,\|\, s \,\|\, e, \mathsf{H.ol}]$
Return $T_H[i,s,X]$

$\mathrm{ROSim}(z,\kappa)$

If $T[z,\kappa] = \bot$ then
  $T[z,\kappa] \leftarrow_\$ \{0,1\}^{\kappa}$
  $hk' \,\|\, i \,\|\, s \,\|\, e \leftarrow z$
  If $(hk',\kappa) = (hk,\mathsf{H.ol})$ then
    $S_{RO} \leftarrow S_{RO} \cup \{(i,s,e)\}$
    For each $(i',s',X) \in S_{\mathsf{ch}}$ do
      If $(i',s') = (i,s)$ and $\mathrm{DH}(X,e)$ then
        abort(i,e)
    For each $((i',s',X) \in S_H$ do
      If $(i',s') = (i,s)$ and $\mathrm{DH}(X,e)$ then
        $T[z,\kappa] \leftarrow T_H[i,s,X]$
Return $T[z,\kappa]$

Figure 1.20. Adversary $\mathcal{B}$ for proof of Lemma 5.

---

before v was incremented to its current value (i.e., $(v,s,g^{x[v]}) \in S'_{\mathsf{hash}}$). Note then, that the query was made before $x[v]$ was sampled. For a query (i,s,X) made while $i < v$, the probability that $X = g^{x[i]}$ is $1/p$. A union bound over the queries to $\textsc{Hash}$ gives the second claim.

Claim (3). Now consider games G_1 and G_2. In G_2 consistency between T and T_{CH} is no longer maintained. They are identical until the flag $\mathsf{bad}_{1,2}$ is set true. By the fundamental lemma of game playing, we have

$$\Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_2] \le \Pr[\mathsf{bad}_{1,2}^{\mathrm{G}_2}].$$

In order for the adversary $\mathcal{O}$ to set $\mathsf{bad}_{1,2}$ true in game G_2 is has to call $\mathrm{RO}(hk \,\|\, i \,\|\, s \,\|\, e, \mathsf{H.ol})$ and $\textsc{Ch}(s)$ such that $(v,s,g^{x[v]\cdot y}) = (i,s,e)$ where v represents the value of that variable when the

call to $\textsc{Ch}(s)$ is made. This means that adversary $\mathcal{O}$ must be able to compute $e = g^{x[v]\cdot y}$ from just $X = g^{x[v]}$ (returned by $\textsc{Up}$) and $Y = g^y$ (passed as input to $\mathcal{O}$). This natural motivates using a reduction to SCDHE.

Consider the adversary $\mathcal{B}$ shown in Fig. 1.20. The special pseudocode command **abort**(v,e) tells $\mathcal{B}$ to immediate halt execution with output (v,e). It simulates oracles $\textsc{Up}$, $\textsc{Exp}$ for adversary $\mathcal{O}$ by using its own oracles with the same names. It simulates oracles $\textsc{Ch}$, $\textsc{Hash}$, and RO by maintaining its own copies of tables $T_{\textsc{Ch}}$, T_H, and T.

However, $\mathcal{B}$ does not know y so it cannot calculate X_{cur}^y or X^y to index into $T_{\textsc{Ch}}$ and T_H the same way that G_2 does. Instead it will index using X_{cur} in $\textsc{ChSim}$ and X in $\textsc{HashSim}$. This is consistent because exponentiation by y is a bijective, but causes difficulty for maintaining consistency with T. For $z = hk \,\|\, i \,\|\, s \,\|\, e$ it be able to verify if $e = X^y$ for any (i,s,X) entry of $T_{\textsc{Ch}}$ or T_H. This issue is resolved via the highlighted code. The set S_{ch} is used together with the new sets S_H and S_{RO} to track the non-$\perp$ entries of tables. When a check for consistency is required $\mathcal{B}$ loops through the appropriate set and uses its DH oracle to check if relations of the form $(e = X^y)$ hold.

Finally, whenever $\mathsf{bad}_{1,2}$ would be set $\mathcal{B}$ aborts and outputs the $g^{x[v]\cdot y}$ it has found. From the preceding DH query we can verify that $\mathcal{B}$ will always wins when it aborts. Note the use of op ensures the tuple it returns satisfies valid. Thus, we have that

$$\Pr[\mathsf{bad}_{1,2}^{\mathrm{G}_2}] = \Pr[\mathrm{SCDHE}_{\mathbb{G}}^{\mathcal{B}}] = \mathsf{Adv}_{\mathbb{G},\mathcal{B}}^{\mathsf{scdhe}}.$$

Note that for each pair of calls that $\mathcal{O}$ makes to RO and $\textsc{Ch}$ (resp. RO and $\textsc{Hash}$), adversary $\mathcal{B}$ will make at most one query to its DH oracle, during the later of the pair of calls. Therefor adversary $\mathcal{B}$ makes at most $q_{\mathrm{RO}} \cdot (q_{\textsc{Ch}} + q_{\textsc{Hash}})$ queries to its DH oracle. The running time of adversary $\mathcal{B}$ is roughly that of $\mathcal{O}$, plus the extra computation that is required to evaluate the $q_{\mathrm{RO}} \cdot (q_{\textsc{Ch}} + q_{\textsc{Hash}})$ queries mentioned above. The other claims about the oracle queries made by $\mathcal{B}$ are easily verified by examining its code.

Claim (4). The proof concludes by verifying $\Pr[\mathrm{G}_2] = \Pr[\mathrm{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H},0}]$. Note that in G_2 the tables T_H and T are kept consistent and so accurately model the random oracle in $\mathrm{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H},0}$. Additionally, the use of table T exactly matches that of mem in $\mathrm{ODHER}^{\mathcal{O}}_{\mathbb{G},\mathsf{H},0}$ because it is decoupled from these other tables. □

Reducing SCDHE to SCDH.

The second intermediate lemma is as follows.

Lemma 6. *Let $\mathbb{G}$ be a cyclic group of order $p \in \mathbb{N}$, and let $\mathbb{G}^*$ denote the set of its generators. Let $\mathcal{B}$ be an adversary attacking the* SCDHE*-security of $\mathbb{G}$ that makes q_{DH} queries to its* DH *oracle and q_{UP} queries to its* UP *oracle. Then there is an adversary $\mathcal{S}$ attacking the* SCDH*-security of $\mathbb{G}$ such that*

$$\mathsf{Adv}^{\mathsf{scdhe}}_{\mathbb{G},\mathcal{B}} \leq q_{\mathrm{UP}} \cdot \mathsf{Adv}^{\mathsf{scdh}}_{\mathbb{G},\mathcal{S}}.$$

Adversary $\mathcal{S}$ makes at most q_{DH} queries to its DH *oracle and its running time is approximately that of $\mathcal{B}$.*

Lemma 6. We build an adversary $\mathcal{S}$ attacking the SCDH-security of $\mathbb{G}$ as follows:

Adversary $\mathcal{S}^{\mathrm{DH}}(g, X_{\mathcal{S}}, Y)$	UPSIM
$j_{\mathcal{S}} \leftarrow_{\$} \{0, \ldots, q_{\mathrm{UP}} - 1\}$; $v \leftarrow -1$ $(j, Z) \leftarrow_{\$} \mathcal{B}^{\mathrm{DHSIM},\mathrm{UPSIM},\mathrm{EXPSIM}}(g, Y)$ Return Z	$v \leftarrow v + 1$; $x \leftarrow_{\$} \mathbb{Z}_p$; $X \leftarrow g^x$ If $(v = j_{\mathcal{S}})$ then $X \leftarrow X_{\mathcal{S}}$ Return X
<u>DHSIM(X, Z)</u>	<u>EXPSIM</u>
Return $\mathrm{DH}(X, Z)$	If $(v = j_{\mathcal{S}})$ then return $\perp$ Return x

Adversary $\mathcal{S}$ simulates game $\mathrm{SCDHE}_{\mathbb{G}}$ for adversary $\mathcal{B}$, answering $\mathcal{B}$'s calls to oracle DH using its own DH oracle (it has the same functionality because $\mathcal{S}$ runs $\mathcal{B}$ with g, Y as input). Adversary $\mathcal{S}$ chooses a random index $j_{\mathcal{S}}$ between 0 and $q_{\mathrm{UP}} - 1$, representing its guess of which index j is going to be returned by adversary $\mathcal{B}$. It then uses $X_{\mathcal{S}}$ as the challenge value to answer $\mathcal{B}$'s call to oracle UP when $v = j_{\mathcal{S}}$. It samples its own challenge exponents for the rest of the UP calls, meaning it is also able to answer $\mathcal{B}$'s calls to EXP whenever $v \neq j_{\mathcal{S}}$. Adversary $\mathcal{S}$ is not

able to properly simulate the expose oracle for $\mathcal{B}$ when $v = j_{\mathcal{S}}$, but this is not important for our reduction because $\mathcal{B}$ could not have won by returning $j = j_{\mathcal{S}}$ after making such EXP call (it would set $\mathsf{op}[j_{\mathcal{S}}] =$ "exp" in game $\mathrm{SCDHE}_{\mathbb{G}}$).

Let $j_{\mathcal{B}}$ denote the index guessed by $\mathcal{S}$ in game $\mathrm{SCDH}_{\mathbb{G}}^{\mathcal{S}}$, and let $j_{\mathcal{B}}$ denote the index returned by $\mathcal{B}$ in game $\mathrm{SCDHE}_{\mathbb{G}}^{\mathcal{B}}$. For any $j \in \{0, 1, \ldots, q_{\mathrm{UP}} - 1\}$ we have

$$\Pr\left[\mathrm{SCDH}_{\mathbb{G}}^{\mathcal{S}} \mid j_{\mathcal{S}} = j\right] \geq \Pr[\mathrm{SCDHE}_{\mathbb{G}}^{\mathcal{B}} \wedge j_{\mathcal{B}} = j].$$

This is an inequality because it is possible that $\mathcal{S}$ wins in $\mathrm{SCDH}_{\mathbb{G}}^{\mathcal{S}}$ by simulating $\mathcal{B}$ that does not win in $\mathrm{SCDHE}_{\mathbb{G}}^{\mathcal{B}}$ (e.g. this happens if $\mathcal{B}$ returns (j, Z) after calling EXP when $v = j$). We get the following:

$$\begin{aligned}
\mathsf{Adv}_{\mathbb{G},\mathcal{S}}^{\mathsf{scdh}} &= \Pr[\mathrm{SCDH}_{\mathbb{G}}^{\mathcal{S}}] \\
&= \sum_{j=0}^{q_{\mathrm{UP}}-1} \Pr[\mathrm{SCDH}_{\mathbb{G}}^{\mathcal{S}} | j_{\mathcal{S}} = j] \cdot \Pr[j_{\mathcal{S}} = j] \\
&\geq \sum_{j=0}^{q_{\mathrm{UP}}-1} \Pr[\mathrm{SCDHE}_{\mathbb{G}}^{\mathcal{B}} \wedge j_{\mathcal{B}} = j] \cdot \frac{1}{q_{\mathrm{UP}}} \\
&= \frac{1}{q_{\mathrm{UP}}} \cdot \Pr[\mathrm{SCDHE}_{\mathbb{G}}^{\mathcal{B}}] \\
&= \frac{1}{q_{\mathrm{UP}}} \cdot \mathsf{Adv}_{\mathbb{G},\mathcal{B}}^{\mathsf{scdhe}}
\end{aligned}$$

The claimed number of oracle queries and running time of $\mathcal{S}$ follows from its construction.

□

Theorem 3. This theorem is a direct consequence of Lemma 5 and Lemma 6. □

1.5.2 SCDHE reduction to SCDH with rewinding

In previous section we provided a reduction from ODHE to SCDH in ROM, by combining two intermediate lemmas – Lemma 5 and Lemma 6. The latter lemma reduced SCDHE to SCDH and incurred a multiplicative loss in advantage. We now use the self-reducibility property of

Diffie-Hellman to state and prove an alternative reduction from SCDHE to SCDH that avoids such multiplicative loss. The new reduction is formalized as Lemma 7 below. Combining Lemma 7 with Lemma 5 from the previous section will yield a more efficient overall reduction from ODHE to SCDH in ROM, as stated in Theorem 4 of Section 1.5.

Outline of the new reduction.

Recall that the SCDH adversary $\mathcal{S}$ used for the proof of Lemma 6 attempts to simulate the provided SCDHE adversary $\mathcal{B}$. Both adversaries take a challenge value $Y = g^y$ as input for some secret uniformly random exponent y. Adversary $\mathcal{S}$ tries to guess which of $\mathcal{B}$'s queries to its oracle UP —returning some group element $X = g^x$— will be used for constructing $\mathcal{B}$'s output value $Z = g^{xy}$. The SCDH adversary then uses the challenge value $X_{\mathcal{S}} = g^{x_{\mathcal{S}}}$ from its own game as the output of the chosen UP query, and answers every other UP query by sampling a uniformly random exponent x itself and returning $X = g^x$.

The new SCDH adversary $\mathcal{S}_u$ for the reduction in this section will instead re-randomize its challenge value $X_{\mathcal{S}_u} = g^{x_{\mathcal{S}_u}}$ in order to answer UP queries for the simulated SCDHE adversary $\mathcal{B}$. To answer an oracle UP query, adversary $\mathcal{S}_u$ will occasionally choose a uniformly random exponent t and return a uniformly random-looking group element $X_{\mathcal{S}_u} \cdot g^t$. If the SCDHE adversary wins its game by returning $Z = (X_{\mathcal{S}_u} \cdot g^t)^y$ then the SCDH adversary can recover $g^{x_{\mathcal{S}_u} y}$ by computing $Z \cdot Y^{-t}$.

However, if adversary $\mathcal{S}_u$ generates its answer to UP query as above, then it cannot answer $\mathcal{B}$'s subsequent query to its EXP oracle, because $\mathcal{S}_u$ does not know the corresponding exponent $x_{\mathcal{S}_u} \cdot t$. To avoid this problem, adversary $\mathcal{S}_u$ for every call to UP guesses whether or not $\mathcal{B}$ will call EXP to recover the corresponding exponent. If adversary $\mathcal{S}_u$ expects a call to EXP, then it answers UP query using $X = g^x$ for a uniformly random exponent x; otherwise, it answers UP query using $X_{\mathcal{S}_u} \cdot g^t$ for a uniformly random exponent t as described above. If $\mathcal{S}_u$ determines that its guess was wrong, it "rewinds" adversary $\mathcal{B}$ back to the corresponding state and attempts to guess again.

Game $\text{SCDHE}_{\mathbb{G}}^{\mathcal{B}}$

$g \leftarrow\!\$\, \mathbb{G}^* \,;\; y \leftarrow\!\$\, \mathbb{Z}_p \,;\; v \leftarrow -1$

$(\text{proc}, \text{args}, \vec{k}, \text{done}) \leftarrow\!\$\, \mathcal{B}_1(g, g^y)$

While not done do

  If proc = "dh" then

    $(X, Z) \leftarrow \text{args} \,;\; \text{out} \leftarrow \text{DH}(X, Z)$

  If proc = "up" then out ← Up

  If proc = "exp" then out ← Exp

  $(\text{proc}, \text{args}, \vec{k}, \text{done}) \leftarrow\!\$\, \mathcal{B}_2(\vec{k}, \text{out})$

$(j, Z) \leftarrow \vec{k}$

Return $(0 \leq j \leq v)$ and $(\text{op}[j] \neq \text{"exp"})$ and $(Z = g^{x[j] \cdot y})$

DH(X,Z)

Return $(X^y = Z)$

Up

$v \leftarrow v + 1 \,;\; x[v] \leftarrow\!\$\, \mathbb{Z}_p \,;$ Return $g^{x[v]}$

Exp

$\text{op}[v] = \text{"exp"}$; Return $x[v]$

Figure 1.21. Game defining SCDHE assumption in group $\mathbb{G}$. This non-black-box definition extends that of Fig. 1.18 to require that adversary $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$ halts after every oracle call.

Alternative definition of SCDHE.

We now redefine the SCDHE security in group $\mathbb{G}$ in a way that allows to "rewind" the adversary playing in this game. Let $\mathbb{G}$ be a cyclic group of order $p \in \mathbb{N}$, and let $\mathbb{G}^*$ denote the set of its generators. Consider game SCDHE of Fig. 1.21, associated to group $\mathbb{G}$ and to an adversary $\mathcal{B} = (\mathcal{B}_1, \mathcal{B}_2)$. As opposed to the original definition from Fig. 1.18, now adversary $\mathcal{B}$ must temporarily halt and output its state whenever it calls one of its oracles. Specifically, adversary $\mathcal{B}$ is required to output a tuple $(\text{proc}, \text{args}, \vec{k}, \text{done})$ where proc is the name of the oracle it wants to call, args is the argument string it wants to pass to that oracle, $\vec{k}$ is the state of $\mathcal{B}$, and done is a boolean flag indicating whether $\mathcal{B}$ has finished playing the game (in which case $\vec{k}$ will be interpreted as its final output). The advantage of $\mathcal{B}$ in breaking the SCDHE security of $\mathbb{G}$ is defined as $\text{Adv}^{\text{scdhe}}_{\mathbb{G},\mathcal{B}} = \Pr[\text{SCDHE}^{\mathcal{B}}_{\mathbb{G}}]$. We emphasize that $\vec{k}$ represents *all* of the state being stored by $\mathcal{B}$ so "rewinding" $\mathcal{B}$ will simply consist of calling $\mathcal{B}$ again on a prior value of $\vec{k}$.

```
Adversary S_u^DH(g, X_{S_u}, Y)                              | Procedure Revert
v ← −1 ; b_g ← 1 ; (proc, args, k⃗, done) ←$ B_1(g, Y)        | maxErr ← maxErr − 1
While not done do                                            | If maxErr < 0 then abort
  If proc = "dh" then                                        | k⃗ ← lastSave
    (X, Z) ← args ; out ← DH(X, Z)                           | goto label
  If proc = "up" then
    If (b_g = 0) and (op[v] ≠ "exp") then Revert
    maxErr ← u ; lastSave ← k⃗ ; v ← v + 1
    //label//
    b_g ←$ {0,1} ; t[v] ←$ Z_p ; out ← X_{S_u}^{b_g} · g^{t[v]}
  If proc = "exp" then
    If (b_g = 1) and (v ≠ −1) then Revert
    op[v] ← "exp" ; out ← t[v]
  (proc, args, k⃗, done) ←$ B_2(k⃗, out)
If (b_g = 0) and (op[v] ≠ "exp") then Revert
(j, Z) ← k⃗ ; Return Z · Y^{−t[j]}
```

Figure 1.22. Adversary $\mathcal{S}_u$ for proof of Lemma 7.

Reducing SCDHE to SCDH using rewinding.

The rewinding-based lemma is as follows.

Lemma 7. *Let $\mathbb{G}$ be a cyclic group of order $p \in \mathbb{N}$, and let $\mathbb{G}^*$ denote the set of its generators. Let $\mathcal{B}$ be an adversary attacking the* SCDHE*-security of $\mathbb{G}$ that makes q_{DH} queries to its* DH *oracle and q_{UP} queries to its* Up *oracle. Let $u \in \mathbb{N}$. Then there is an adversary $\mathcal{S}_u$ attacking the* SCDH*-security of $\mathbb{G}$ such that*

$$\mathsf{Adv}^{\mathsf{scdhe}}_{\mathbb{G},\mathcal{B}} \leq \mathsf{Adv}^{\mathsf{scdh}}_{\mathbb{G},\mathcal{S}_u} + q_{\mathrm{UP}} \cdot 2^{-u}.$$

Let $q^* = 1 + u \cdot q_{\mathrm{UP}}$. Adversary $\mathcal{S}_u$ makes at most $q^* \cdot q_{\mathrm{DH}}$ queries to its DH *oracle. Its running time is approximately q^* times that of $\mathcal{B}$.*

The expected number of oracle queries and running time of adversary $\mathcal{S}_u$ are better than the worst-case guarentees provided above. The expected number of oracle queries to DH made by adversary $\mathcal{S}_u$ is about twice that of $\mathcal{B}$. The expected running time of adversary $\mathcal{S}_u$ is approximately twice that of $\mathcal{B}$.

Lemma 7. We build an adversary $\mathcal{S}_u$ attacking the SCDH-security in $\mathbb{G}$, as defined in Fig. 1.22.

We use **abort** as a shorthand for $\mathcal{S}_u$ halting execution and returning an output value (j,Z) that guarantess that $\mathcal{S}_u$ loses in game $\mathrm{SCDH}_{\mathbb{G}}^{\mathcal{S}_u}$, for example $(-1,g)$. As a part of adversary $\mathcal{S}_u$, we define an auxiliary procedure REVERT that contains the code for rewinding the simulated adversary $\mathcal{B}$. It holds no special meaning in our security proof; its only purpose is to avoid writing duplicate code at every position that calls rewinding, essentially simplifying the definition of $\mathcal{S}_u$.

Adversary $\mathcal{S}_u$ simulates the $\mathrm{SCDHE}_{\mathbb{G}}$ game for adversary $\mathcal{B}$, according to the definition of SCDHE in Fig. 1.21. Every time adversary $\mathcal{B}$ makes a call to oracle UP, adversary $\mathcal{S}_u$ samples a random bit b_g to determine how to answer this query. If $b_g = 1$ then UP will return $X_{\mathcal{S}_u} \cdot g^{t[v]}$ for a uniformly random exponent $t[v] \in \mathbb{Z}_p$. If $b_g = 0$ then UP will return just $g^{t[v]}$ for a uniformly random exponent $t[v] \in \mathbb{Z}_p$. Furthermore, it is assumed that $\mathcal{B}$ will call oracle EXP prior to making its next query to oracle UP if and only if $b_g = 0$.

The goal here is that adversary $\mathcal{S}_u$ must be capable of answering EXP query whenever this query is made. Note that $\mathcal{S}_u$ can only answer it if $b_g = 0$, unless it can compute the discrete logarithm of $X_{\mathcal{S}_u}$ in group $\mathbb{G}$. On the other hand, for any challenge value returned by UP that was not exposed, adversary $\mathcal{S}_u$ must ensure that it was generated as $X_{\mathcal{S}_u} \cdot g^{t[v]}$ for a known value $t[v]$. This guarantees that if $\mathcal{B}$ wins in game $\mathrm{SCDHE}_{\mathbb{G}}^{\mathcal{B}}$ by returning some (j,Z) such that $Z = X_{\mathcal{S}_u}^{y} \cdot g^{t[j] \cdot y}$, then adversary $\mathcal{S}_u$ will always be able to compute $Z \cdot Y^{-t[j]} = X_{\mathcal{S}_u}^{y}$ to win in game $\mathrm{SCDH}_{\mathbb{G}}^{\mathcal{S}_u}$.

Every time adversary $\mathcal{B}$ makes a call to oracle UP, adversary $\mathcal{S}_u$ checks whether its previous guess about $\mathcal{B}$'s behavior (as reflected by b_g) was correct. If the guess was correct, then $\mathcal{S}_u$ saves $\mathcal{B}$'s current state $\vec{k}$ in variable lastSave and proceeds to make a new guess about $\mathcal{B}$'s future behavior. If the guess was wrong, then $\mathcal{S}_u$ "rewinds" adversary $\mathcal{B}$ back to the state it had during its previous call to oracle UP, by restoring $\mathcal{B}$'s state from the current value of lastSave and calling **goto label** to move the instruction pointer to **//label//**; adversary $\mathcal{S}_u$ then makes a new attempt to guess $\mathcal{B}$'s behavior, sampling a random bit b_g. We ensure that $\mathcal{S}_u$ eventually halts by allowing it to make at most maxErr wrong guesses in a row. If $\mathcal{S}_u$ exceeds this number of

```
Games G_0–G_1
g ←$ G* ; y ←$ Z_p ; x ←$ Z_p ; v ← −1 ; b_g ← 1
(proc, args, k⃗, done) ←$ B_1(g, g^y)
While not done do
   If proc = "dh" then
      (X, Z) ← args ; out ← (X^y = Z)
   If proc = "up" then
      If (b_g = 0) and (op[v] ≠ "exp") then REVERT
      maxErr ← u ; lastSave ← k⃗ ; v ← v + 1
      //label//
      b_g ←$ {0,1} ; t[v] ←$ Z_p ; out ← g^{x·b_g} · g^{t[v]}
   If proc = "exp" then
      If (b_g = 1) and (v ≠ −1) then REVERT
      op[v] ← "exp" ; out ← t[v]
   (proc, args, k⃗, done) ←$ B_2(k⃗, out)
If (b_g = 0) and (op[v] ≠ "exp") then REVERT
(j, Z) ← k⃗
Return (Z · g^{−t[j]·y} = g^{xy})

Procedure REVERT
maxErr ← maxErr − 1
If maxErr < 0 then
   bad_v ← true
   abort                                         // G_0
(v, k⃗) ← lastSave
goto label
```

Figure 1.23. Games G_0, G_1 for proof of Lemma 7.

---

wrong guesses, it immediately calls **abort** to halt its execution.

We now analyze the success probability of adversary $\mathcal{S}_u$. Consider games G_0, G_1 of Fig. 1.23. Lines not annotated with comments are common to both games. Game G_0 is equivalent to $\mathrm{SCDH}_{\mathbb{G}}^{\mathcal{S}_u}$, with the code of $\mathcal{S}$ inserted, and simplified to reference g^x instead of $X_{\mathcal{S}_u}$. It follows that

$$\Pr[G_0] = \Pr[\mathrm{SCDH}_{\mathbb{G}}^{\mathcal{S}_u}]. \tag{1.6}$$

Game G_1 is equivalent to G_0, except that it never calls **abort**. Let $\Pr[\mathsf{bad}^{G_0}]$ denote the probability that a bad_v flag is set true in game G_0 for some $v \in \{0, 1, \ldots, q_{\mathrm{UP}} - 1\}$. Note that **abort** is only called when this happens, and the two games are identical-until-bad. According to the

fundamental lemma of game playing [16], we have

$$\Pr[\mathrm{G}_1] \leq \Pr[\mathrm{G}_0] + \Pr[\mathsf{bad}^{\mathrm{G}_0}]. \tag{1.7}$$

Note that adversary $\mathcal{B}$ always gets uniformly random group elements in response to its oracle UP queries, regardless of the value of b_g. Meaning that regardless of the behavior of $\mathcal{B}$ the probability that b_g is chosen correctly at any point is exactly $1/2$. It follows that the output distribution of $\mathcal{B}$ induced in $\Pr[\mathrm{G}_1]$ is exactly that same as that induced in $\mathrm{SCDHE}_{\mathbb{G}}^{\mathcal{B}}$. It follows that the condition $Z \cdot g^{-t[j]\cdot y} = g^{xy}$ in game $\Pr[\mathrm{G}_1]$ will hold whenever (j, Z) would have been a correct answer in game SCDHE. This gives us

$$\Pr[\mathrm{SCDHE}_{\mathbb{G}}^{\mathcal{B}}] \leq \Pr[\mathrm{G}_1]. \tag{1.8}$$

Finally, we now bound the probability that a bad flag is set in G_0 by the following inequalities.

$$\Pr[\mathsf{bad}^{\mathrm{G}_0}] \leq \sum_{v=0}^{q_{\mathrm{UP}}-1} \Pr[\mathsf{bad}_v^{\mathrm{G}_0}] \leq \sum_{v=0}^{q_{\mathrm{UP}}-1} 2^{-u} = q_{\mathrm{UP}} \cdot 2^{-u}. \tag{1.9}$$

The first inequality comes from a simple union bound. To see the second inequality note that for any particular bad_v to be set true the guess bit b_g must be incorrect u times in a row, which happens with probability 2^{-u}.

Finally, the theorem statement follows from Equations (1.6)–(1.9):

$$\Pr[\mathrm{SCDHE}_{\mathbb{G}}^{\mathcal{B}}] \leq \Pr[\mathrm{SCDH}_{\mathbb{G}}^{\mathcal{S}_u}] + q_{\mathrm{UP}} \cdot 2^{-u}.$$

The claimed number of oracle queries and the worst-case running time of $\mathcal{S}_u$ follows from the fact that $\mathcal{S}_u$ will run $\mathcal{B}$ once, and depending on $\mathcal{B}$'s behavior it will rewind $\mathcal{B}$ at most $u \cdot q_{\mathrm{UP}}$ times. In the worst case, each rewinding will reset $\mathcal{B}$ back to the very beginning of its execution and it would have to be run all over again. □

Theorem 4. This theorem is a direct consequence of Lemma 5 and Lemma 7. □

Game $\text{UFORGE}_{\mathsf{RKE}}^{\mathcal{M}}$
$(k, sek_s, (stk_s, stk_r, sek_r)) \leftarrow\$ \mathsf{RKE.IKg}$
$k_s \leftarrow k\,;\; k_r \leftarrow k\,;\; upd \leftarrow\$ \mathcal{M}^{\text{RATCHET}}(stk_s)$
$(sek_r, k_r, acc) \leftarrow\$ \mathsf{RKE.RKg}(stk_s, stk_r, sek_r, upd, k_r)$
Return acc

RATCHET
$prev \leftarrow (r, sek_s, k_s)\,;\; r \leftarrow\$ \mathsf{RKE.RS}$
$(sek_s, k_s, upd) \leftarrow \mathsf{RKE.SKg}(stk_s, sek_s; r)$
$(sek_r, k_r, acc) \leftarrow\$ \mathsf{RKE.RKg}(stk_s, stk_r, sek_r, upd, k_r)$
Return $(prev, upd)$

Figure 1.24. Game defining update unforgeability of ratcheted key exchange scheme RKE.

1.6 Necessity of authenticaticating the update information

In this section we show that if an attacker can forge update information upd for a ratcheted key exchange scheme, then it can be used to break the KIND security of this scheme. Here, a *forged* update information is one that was not produced by the sender, but would still be accepted by the receiver. This result could equivalently be stated as reducing the UFORGE security (that we are about to formally define) of a ratcheted key exchange scheme to its KIND security, meaning that UFORGE is a property that is a necessary condition for KIND security.

Update unforgeability.

Informally, a ratcheted key exchange scheme is secure against update forgeries if an adversary, given access to several samples of update information and to all sender's secrets prior to the generation of these samples, cannot generate its own update information that will be accepted by the receiver. Conversely, an adversary is good at update forgeries if it can do this.

Consider game UFORGE of Fig. 1.24, associated to a ratcheted key exchange scheme RKE and an adversary $\mathcal{M}$. The advantage of $\mathcal{M}$ at breaking the UFORGE security of RKE is given by $\mathsf{Adv}^{\mathsf{uforge}}_{\mathsf{RKE},\mathcal{M}} = \Pr[\text{UFORGE}^{\mathcal{M}}_{\mathsf{RKE}}]$.

Adversary $\mathcal{M}$ is given the sender's static key and is provided with an access to an oracle RATCHET that performs a ratcheted key update for both the sender and the receiver. Oracle

RATCHET returns the sender's secrets from before ratcheting the keys, along with the update information upd that was used for ratcheting receiver's key. Adversary $\mathcal{M}$ can call its oracle an arbitrary number of times, and its goal is to eventually generate its own update information upd that is accepted by the receiver. Note that the first call to oracle RATCHET returns (r, sek_s, k_s) where r is not explicitly initialized; according to our notation conventions from Section 1.1, it is implicitly assumed to be initialized with 0.

Breaking KIND using an update forgery.

We now show that any adversary $\mathcal{M}$ that is successful at attacking the UFORGE security of a ratcheted key exchange scheme RKE can be used to build an adversary $\mathcal{D}$ that is successful at attacking the KIND security of RKE.

Theorem 8. *Let* RKE *be a ratcheted key exchange scheme. Let $\mathcal{M}$ be an adversary attacking the* UFORGE-*security of* RKE *that makes q_{RATCHET} queries to its* RATCHET *oracle. Then there is an adversary $\mathcal{D}$ attacking the* KIND-*security of* RKE *such that*

$$\mathsf{Adv}^{\mathsf{kind}}_{\mathsf{RKE},\mathcal{D}} = \left(\frac{1}{2} - \frac{1}{2^{\mathsf{RKE.kl}+1}}\right) \cdot \mathsf{Adv}^{\mathsf{uforge}}_{\mathsf{RKE},\mathcal{M}}.$$

Adversary $\mathcal{D}$ makes at most $q_{\text{RATCHET}} + 1$ queries to each of its RATSEND*,* RATREC*, and* EXP *oracles. It makes one query to each of its* CHSEND *and* CHREC *oracles. Its running time is approximately that of $\mathcal{M}$.*

The starting idea for the KIND adversary $\mathcal{D}$ is to cause the receiver to update its keys with update information upd that was not generated by the sender. It does that by calling its oracle RATREC on update information upd that it learns from UFORGE adversary $\mathcal{M}$. Then (after another call to RATSEND) oracles CHSEND and CHREC will presumably return differing keys in the "real world" ($b = 1$) and matching keys in the "random world" ($b = 0$). Comparing these keys gives a straightforward attack. We note that this attack does not require $\mathcal{D}$ to call EXP right before calling oracles RATSEND, CHSEND, and CHREC. Hence the restricted flag is false at the time when these challenge oracles are called.

Unfortunately we cannot prove that this attack —as outlined above— is successful for an arbitrary RKE scheme. The difficulty stems from the fact that RKE will not necessarily generate non-matching values in the "real world". For "natural" ratcheted key exchange schemes it seems highly unlikely that keys generated by the two challenge oracles in the "real world" would match. However, rigorously proving this claim is hard.

We will augment the attack idea described above to call EXP with probability $1/2$ prior to calling oracles RATSEND, CHSEND, and CHREC. When EXP is called prior to this sequence of calls, game $\text{KIND}^{\mathcal{D}}_{\mathsf{RKE}}$ sets restricted to true. Depending on whether EXP is called, adversary $\mathcal{D}$ is also defined to flip its output bit.

In the "real world" $(b = 1)$ of game $\text{KIND}^{\mathcal{D}}_{\mathsf{RKE}}$, the output of the challenge oracles is the same regardless of whether restricted is true. So when $b = 1$ the interaction of $\mathcal{D}$ with its challenge oracles in game $\text{KIND}^{\mathcal{D}}_{\mathsf{RKE}}$ does not change depending on whether EXP was called (which happens with probability $1/2$), but its output bit is nonetheless flipped. In this case adversary $\mathcal{D}$ will succeed to guess the challenge bit (meaning it will return $b' = 1$) with probability $1/2$.

In constract, calling EXP in the "random world" $(b = 0)$ of game $\text{KIND}^{\mathcal{D}}_{\mathsf{RKE}}$ results in the challenge oracles returning outputs that have different equality patterns. If EXP was called, then restricted is true and adversary will get different keys with high probability in the "random world", because CHSEND returns a uniformly random key and CHREC returns a real key. If EXP was not called, then restricted is false and adversary will always get same keys from the challenge oracles in the "random world". By flipping adversary's output value in one of these two cases, we get an adversary that guesses the correct challenge bit (meaning it returns $b' = 0$) with probability close to 1. This is sufficient to distinguish between the "real world" and the "random world".

Theorem 8. We build an adversary $\mathcal{D}$ attacking the KIND-security of RKE as follows:

Adversary $\mathcal{D}^{\text{RatSend},\text{RatRec},\text{Exp},\text{ChSend},\text{ChRec}}(stk_s)$	RatchetSim
$upd \leftarrow\$\ \mathcal{M}^{\text{RatchetSim}}(stk_s)$; $b_{\mathcal{D}} \leftarrow\$\ \{0,1\}$	$prev \leftarrow \text{Exp}$
If $(b_{\mathcal{D}} = 1)$ then Exp	$upd \leftarrow \text{RatSend}$
$acc_{\mathcal{D}} \leftarrow \text{RatRec}(upd)$	$\text{RatRec}(upd)$
If not $acc_{\mathcal{D}}$ then return 1	Return $(prev, upd)$
RatSend ; $k_s \leftarrow \text{ChSend}$; $k_r \leftarrow \text{ChRec}$	
If $(k_s = k_r)$ then return $b_{\mathcal{D}}$ else return $1 - b_{\mathcal{D}}$	

Adversary $\mathcal{D}$ first uses adversary $\mathcal{M}$ to generate forged update information upd (the simulation of $\mathcal{M}$'s oracle RatchetSim is straightforward). Then $\mathcal{D}$ samples a bit $b_{\mathcal{D}}$ to decide whether it should make a call to Exp before proceeding. It then calls RatRec with update information upd to derive key k_r (immediately returning 1 if this update fails), calls RatSend to derive what would have been the "corresponding" key k_s, and finally calls both challenge oracles to get k_s and k_r. If $k_s = k_r$ then $\mathcal{D}$ returns $b_{\mathcal{D}}$ as its output, otherwise it returns $1 - b_{\mathcal{D}}$.

Consider game $\text{KIND}^{\mathcal{D}}_{\mathsf{RKE}}$. Let b denote the challenge bit in this game. Let $b_{\mathcal{D}}$ and $acc_{\mathcal{D}}$ be the values computed by adversary $\mathcal{D}$ playing in this game. Then the following will hold:

$$\Pr[\text{KIND}^{\mathcal{D}}_{\mathsf{RKE}} \,|\, b = 1, b_{\mathcal{D}} = 1, acc_{\mathcal{D}}] = 1 - \Pr[\text{KIND}^{\mathcal{D}}_{\mathsf{RKE}} \,|\, b = 1, b_{\mathcal{D}} = 0, acc_{\mathcal{D}}], \tag{1.10}$$

$$\Pr[\text{KIND}^{\mathcal{D}}_{\mathsf{RKE}} \,|\, b = 0, b_{\mathcal{D}} = 1, acc_{\mathcal{D}}] = 1 - 2^{-\mathsf{RKE.kl}}, \tag{1.11}$$

$$\Pr[\text{KIND}^{\mathcal{D}}_{\mathsf{RKE}} \,|\, b = 0, b_{\mathcal{D}} = 0, acc_{\mathcal{D}}] = 1. \tag{1.12}$$

We now justify each of these equations. For this purpose, let k_s and k_r be the values computed by adversary $\mathcal{D}$ playing in game $\text{KIND}^{\mathcal{D}}_{\mathsf{RKE}}$.

To justify Equation (1.10), note that if $b = 1$ then the values of k_s and k_r that are compared by $\mathcal{D}$ will both be generated by RKE in game $\text{KIND}^{\mathcal{D}}_{\mathsf{RKE}}$, regardless of whether Exp has been called. Thus the probability of k_s and k_r being equal does not depend on the bit $b_{\mathcal{D}}$. Assume that $b = 1$ and $acc_{\mathcal{D}} = \mathsf{true}$. If $b_{\mathcal{D}} = 1$ then $\mathcal{D}$ returns 1 (the correct value of b) whenever $k_s = k_r$. If $b_{\mathcal{D}} = 0$ then $\mathcal{D}$ returns 1 whenever $k_s \neq k_r$. This implies the stated equality.

When $b = 0$ and $b_{\mathcal{D}} = 1$, the value of k_s will have been chosen uniformly at random, but k_r will have been generated by RKE because the restricted flag will be set to true. Thus the

probability of them being equal is $2^{-\mathsf{RKE.kl}}$. Assume that $b=0$ and $acc_{\mathcal{D}} = \mathsf{true}$. If $b_{\mathcal{D}} = 1$ then $\mathcal{D}$ returns 0 (the correct value of b) whenever $k_s \neq k_r$. This implies Equation (1.11).

When $b=0$ and $b_{\mathcal{D}} = 0$, the restricted flag will be false, so game $\mathrm{KIND}^{\mathcal{D}}_{\mathsf{RKE}}$ forces k_s and k_r to be the same random value. Assume that $b=0$ and $acc_{\mathcal{D}} = \mathsf{true}$. If $b_{\mathcal{D}} = 0$ then $\mathcal{D}$ returns 0 (the correct value of b) whenever $k_s = k_r$. This implies Equation (1.12).

We now use Equations (1.10)–(1.12) to compute the conditional probability that $\mathcal{D}$ wins in game $\mathrm{KIND}^{\mathcal{D}}_{\mathsf{RKE}}$, given that $acc_{\mathcal{D}} = \mathsf{true}$. The result is as follows:

$$\begin{aligned}
\Pr[\mathrm{KIND}^{\mathcal{D}}_{\mathsf{RKE}} \,|\, acc_{\mathcal{D}}] &= \sum_{c\in\{0,1\}} \sum_{d\in\{0,1\}} \Pr[\mathrm{KIND}^{\mathcal{D}}_{\mathsf{RKE}} \,|\, b=c, b_{\mathcal{D}}=d, acc_{\mathcal{D}}] \cdot \frac{1}{4} \\
&= \frac{1}{4} \cdot \left(1 + \left(1 - 2^{-\mathsf{RKE.kl}}\right) + 1\right) \\
&= \frac{1}{4} \cdot \left(3 - 2^{-\mathsf{RKE.kl}}\right). \qquad (1.13)
\end{aligned}$$

Note that adversary $\mathcal{D}$ setting $acc_{\mathcal{D}} = \mathsf{true}$ while playing in game $\mathrm{KIND}^{\mathcal{D}}_{\mathsf{RKE}}$ is equivalent to adversary $\mathcal{M}$ winning in game $\mathrm{UFORGE}^{\mathcal{M}}_{\mathsf{RKE}}$. It follows that

$$\Pr[acc_{\mathcal{D}}] = \mathsf{Adv}^{\mathsf{uforge}}_{\mathsf{RKE},\mathcal{M}}. \qquad (1.14)$$

Furthermore, if $acc_{\mathcal{D}} = \mathsf{false}$ then adversary $\mathcal{D}$ wins in game $\mathrm{KIND}^{\mathcal{D}}_{\mathsf{RKE}}$ with probability $1/2$ because (prior to halting) it only queries oracles $\textsc{RatSend}$, $\textsc{RatRec}$, $\textsc{Exp}$ that behave independently of the challenge bit in game $\mathrm{KIND}^{\mathcal{D}}_{\mathsf{RKE}}$. We have

$$\Pr[\mathrm{KIND}^{\mathcal{D}}_{\mathsf{RKE}} \,|\, \neg acc_{\mathcal{D}}] = \frac{1}{2}. \qquad (1.15)$$

Finally, we compute the advantage of adversary $\mathcal{D}$ from Equations (1.13)–(1.15) as

follows:

$$
\begin{aligned}
\mathsf{Adv}^{\mathsf{kind}}_{\mathsf{RKE},\mathcal{D}} &= 2 \cdot \Pr[\mathrm{KIND}^{\mathcal{D}}_{\mathsf{RKE}}] - 1 \\
&= 2 \cdot \Pr[\,\mathrm{KIND}^{\mathcal{D}}_{\mathsf{RKE}} \,|\, acc_{\mathcal{D}}\,] \cdot \Pr[acc_{\mathcal{D}}] \\
&+ 2 \cdot \Pr[\,\mathrm{KIND}^{\mathcal{D}}_{\mathsf{RKE}} \,|\, \neg acc_{\mathcal{D}}\,] \cdot \Pr[\neg acc_{\mathcal{D}}] - 1 \\
&= 2 \cdot \frac{1}{4} \cdot \left(3 - 2^{-\mathsf{RKE.kl}}\right) \cdot \mathsf{Adv}^{\mathsf{uforge}}_{\mathsf{RKE},\mathcal{M}} + 2 \cdot \frac{1}{2} \cdot (1 - \mathsf{Adv}^{\mathsf{uforge}}_{\mathsf{RKE},\mathcal{M}}) - 1 \\
&= \left(\frac{1}{2} - \frac{1}{2^{\mathsf{RKE.kl}+1}}\right) \cdot \mathsf{Adv}^{\mathsf{uforge}}_{\mathsf{RKE},\mathcal{M}}.
\end{aligned}
$$

The number of oracle queries and the running time that were claimed in the theorem statement follow from the construction of adversary $\mathcal{D}$. □

1.7 Acknowledgements

We thank the EUROCRYPT 2017 and CRYPTO 2017 reviewers for their comments.

Chapter 1, in full, is a reprint of the material as it appears in Advances in Cryptology - CRYPTO 2017. Bellare, Mihir; Camper Singh, Asha; Jaeger, Joseph; Nyayapati, Maya; Stepanovs, Igors, Springer Lecture Notes in Computer Science volume 10403, 2017. The dissertation author was the primary investigator and author of this paper. The dissertation author was supported in part by NSF grants CNS-1526801 and CNS-1228890

Chapter 2

Optimal Channel Security Against Fine-Grained State Compromise

It is commonly agreed in the cryptography and security community that the Signal protocol is secure. However, the protocol was designed without an explicitly defined security notion. This raises the questions: what security does it achieve and could we do better?

In this chapter we study the latter question, aiming to understand the best possible security of two-party communication in the face of state exfiltration. We formally define this notion of security and design a scheme that provably achieves it.

Security against compromise.

When a party's secret state is exposed we would like both that the security of past messages and (as soon as possible) the security of future messages not be damaged. These notions have been considered in a variety of contexts with differing terminology. The systemization of knowledge paper on secure messaging [64] by Unger, Dechand, Bonneau, Fahl, Perl, Goldberg, and Smith evaluates and systematizes a number of secure messaging systems. In it they describe a variety of terms for these types of security including "forward secrecy," "backwards secrecy," "self-healing," and "future secrecy" and note that they are "controversial and vague." Cohn-Gordon, Cremers, and Garratt [28] study the future direction under the term of post-compromise security and similarly discuss the terms "future secrecy," "healing," and "bootstrapping" and note that they are "intuitive" but "not well-defined." Our security notion intuitively captures any

of these informal terms, but we avoid using any of them directly by aiming generically for the best possible security against compromise.

Channels.

The standard model for studying secure two party communication is that of the (cryptographic) channel. The first attempts to consider the secure channel as a cryptographic object were made by Shoup [61] and Canetti [24]. It was then formalized by Canetti and Krawczyk [26] as a modular way to combine a key exchange protocol with authenticated encryption, which covers both privacy and integrity. Krawczyk [46] and Namprempre [51] study what are the necessary and sufficient security notions to build a secure channel from these primitives.

Modern definitions of channels often draw from the game-based notion of security for stateful authenticated-encryption as defined by Bellare, Kohno, and Namprempre [8]. We follow this convention which assumes initial generation of keys is trusted. In addition to requiring that a channel provides integrity and privacy of the encrypted data, we will require integrity for associated data as introduced by Rogaway [57].

Recently Marson and Poettering [49] closed a gap in the modeling of two-party communication by capturing the bidirectional nature of practical channels in their definitions. We work with their notion of bidirectional channels because it closely models the behavior desired in practice and the bidirectional nature of communication allows us to achieve a fine-grained security against compromise.

Definitional contributions.

This chapter aims to specify and achieve the best possible security of a bidirectional channel against state compromise. We provide a formal, game-based definition of security and a construction that provably achieves it. We analyze our construction in a concrete security framework [5] and give precise bounds on the advantage of an attacker.

To derive the best possible notion of security against state compromise we first specify a basic input-output interface via a game that describes how the adversary interacts with the

channel. This corresponds roughly to combining the integrity and confidentiality games of [49] and adding an oracle that returns the secret state of a specified user to the adversary. Then we specify several attacks that break the security of *any* channel. We define our final security notion by minimally extending the initial interface game to disallow these unavoidable attacks while allowing all other behaviors. Our security definition is consequently the best possible with respect to the specified interface because our attacks rule out the possibility of any stronger notion.

One security notion is an all-in-one notion in the style of [59] that simultaneously requires integrity and privacy of the channel. It asks for the maximal possible security in the face of the exposure of either party's state. A surprising requirement of our definition is that given the state of a user the adversary should not be able to decrypt ciphertexts sent by that user or send forged ciphertexts to that user.

Protocols that update their keys.

The OTR (Off-the-Record) messaging protocol [22] is an important predecessor to Signal. It has parties repeatedly exchange Diffie-Hellman elements to derive new keys. The Double Ratchet Algorithm of Signal uses a similar Diffie-Hellman update mechanism and extends it by using a symmetric key-derivation function to update keys when there is no Diffie-Hellman update available. Both methods of updating keys are often referred to as ratcheting (a term introduced by Langley [47]). While the Double Ratchet Algorithm was explicitly designed to achieve strong notions of security against state compromise with respect to privacy, the designers explicitly consider security against a passive eavesdropper [36]; authenticity in the face of compromise is out of scope.

The first academic security analysis of Signal was due to Cohn-Gordan, Cremers, Dowling, Garratt, and Stebila [29]. They only considered the security of the key exchange underlying the Double Ratchet Algorithm and used a security definition explicitly tailored to understanding its security instead of being widely applicable to any scheme.

In Chapter 1 we sought to formally understand ratcheting as an independent primitive, introducing the notions of (one-directional) ratcheted key exchange and ratcheted encryption. In our model a compromise of the receiving party's secrets permanently and irrevocably disrupts all security, past and future. Further we strictly separate the exchange of key update information from the exchange of messages. Such a model cannot capture a protocol like the Double Ratchet Algorithm for which the two are inextricably combined. On the positive side, we did explicitly model authenticity in the face of compromise.

In [43], Günther and Mazaheri study a key update mechanism introduced in TLS 1.3. Their security definition treats update messages as being out-of-band and thus implicitly authenticated. Their definition is clearly tailored to understand TLS 1.3 specifically.

Instead of analyzing an existing scheme, we strive to understand the best possible security with respect to both privacy and authenticity in the face of state compromise. The techniques we use to achieve this differ from those underlying the schemes discussed above, because all of them rely on exchanging information to create a shared symmetric key that is ultimately used for encryption. Our security notion is not achievable by a scheme of this form and instead requires that asymmetric primitives be used throughout.

Consequently, our scheme is more computationally intensive than those mentioned above. However, as a part of OTR or the Double Ratchet Algorithm, when users are actively sending messages back and forth (the case where efficiency is most relevant), they will be performing asymmetric Diffie-Hellman based key updates prior to most message encryptions. This indicates that the overhead of extra computation with asymmetric techniques is not debilitating in our motivating context of secure messaging. However, the asymmetric techniques we require are likely less efficient than Diffie-Hellman computations so we do not currently know whether our scheme meets realistic efficiency requirements.

Our construction.

Our construction of a secure channel is given in Section 2.5.1. It shows how to generically build the channel from a collision-resistant hash function, a public-key encryption scheme, and a digital signature scheme. The latter two require new versions of the primitives that we describe momentarily.

The hash function is used to store transcripts of the communication in the form of hashes of all sent or received ciphertexts. These transcripts are included as part of every ciphertext and a user will not accept a ciphertext with transcripts that do not match those it has stored locally. Every ciphertext sent by a user is signed by their current digital signature signing key and includes the verification key corresponding to their next signing key. Similarly a user will include a new encryption key with every ciphertext they send. The sending user will use the most recent encryption key they have received from the other user and the receiving user will delete all decryption keys that are older than the one most recently used by the sender.

New notions of public-key encryption and digital signatures.

Our construction uses new forms of public-key encryption and digital signatures that update their keys over time, which we define in Section 2.2. They both include extra algorithms that allow the keys to be updated with respect to an arbitrary string. We refer to them as key-updatable public-key encryption and key-updatable digital signature schemes. In our secure channel construction a user updates their signing key and every decryption key they currently have with their transcript every time they receive a ciphertext.

For public-key encryption we consider encryption with labels and require an IND-CCA style security be maintained even if the adversary is given the decryption key as long as the sequence of strings used to update it is not a prefix of the sequence of strings used to update the encryption key before any challenge queries. We show that such a scheme is can be obtained directly from hierarchical identity-based encryption [40].

For digital signatures, security requires that an adversary is unable to forge a signature

even given the signing key as long as the sequence of strings used to update it is not a prefix of the sequence of strings used to update the verification key. We additionally require that the scheme has unique signatures (i.e. for any sequence of updates and any message an adversary can only find one signature that will verify). We show how to construct this from a digital signature scheme that is forward secure [9] and has unique signatures.

Related work.

Several works [39, 21] extended the definitions of channels to address the stream-based interface provided by channels like TLS, SSH, and QUIC. Our primary motivation is to build a channel for messaging where an atomic interface for messages is more appropriate.

Numerous areas of research within cryptography are motivated by the threat of key compromise. These include key-insulated cryptography [32, 33, 34], secret sharing [60, 50, 63], threshold cryptography [30], proactive cryptography [55], and forward security [42, 31]. Forward security, in particular, was introduced in the context of key-exchange [42, 31] but has since been considered for a variety of primitives including symmetric [19] and asymmetric encryption [25] and digital signature schemes [9]. Green and Miers [41] propose using puncturable encryption for forward secure asynchronous messaging.

In concurrent and independent work, Poettering and Rösler [56] extend the definitions of ratcheted key exchange from [17] to be bidirectional. Their security definition is conceptually similar to our definition for bidirectional channels because both works aim to achieve strong notions of security against an adversary that can arbitrarily and repeatedly learn the secret state of either communicating party. In constructing a secure ratcheted key exchange scheme they make use of a key-updatable key encapsulation mechanism (KEM), a new primitive they introduce in their work. The key-updatable nature of this is conceptually similar to that of the key-updatable public-key encryption and digital signature schemes we introduce in our work. To construct such a KEM they make use of hierarchical identity-based encryption in a manner similar to how we construct key-updatable public-key encryption. The goal of their work differs from ours; they

only consider security for the exchange of symmetric keys while we do so for the exchange of messages.

2.1 Preliminaries

Notation and conventions.

Let $\mathbb{N} = \{0,1,2,\ldots\}$ be the set of non-negative integers. Let ε denote the empty string. If $x \in \{0,1\}^*$ is a string then $|x|$ denotes its length. If X is a finite set, we let $x \leftarrow\!\$\, X$ denote picking an element of X uniformly at random and assigning it to x. By $(X)^n$ we denote the n-ary Cartesian product of X. We let $x_1 \leftarrow x_2 \leftarrow \cdots \leftarrow x_n \leftarrow v$ denote assigning the value v to each variable x_i for $i = 1,\ldots,n$.

If **mem** is a table, we use $\mathbf{mem}[p]$ to denote the element of the table that is indexed by p. By $\mathbf{mem}[0,\ldots,\infty] \leftarrow v$ we denote initializing all elements of **mem** to v. For $a,b \in \mathbb{N}$ we let $v \leftarrow \mathbf{mem}[a,\ldots,b]$ denote setting v equal to the tuple obtained by removing all $\perp$ elements from $(\mathbf{mem}[a],\mathbf{mem}[a+1],\ldots,\mathbf{mem}[b])$. It is the empty vector $()$ if all of these table entries are $\perp$ or if $a > b$. A tuple $\vec{x} = (x_1,\ldots)$ specifies a uniquely decodable concatenation of strings $x_1,\ldots$. We say $\vec{x} \sqsubseteq \vec{y}$ if $\vec{x}$ is a prefix of $\vec{y}$. More formally, $(x_1,\ldots,x_n) \sqsubseteq (y_1,\ldots,y_m)$ if $n \leq m$ and $x_i = y_i$ for all $i \in \{1,\ldots,n\}$. If $\vec{x} = (x_1,\ldots,x_n)$ is a vector and $x \in \{0,1\}^*$ then we define the concatenation of $\vec{x}$ and x to be $\vec{x} \,\|\, x = (x_1,\ldots,x_n,x)$.

Algorithms may be randomized unless otherwise indicated. Running time is worst case. If A is an algorithm, we let $y \leftarrow A(x_1,\ldots;r)$ denote running A with random coins r on inputs $x_1,\ldots$ and assigning the output to y. Any state maintained by an algorithm will explicitly be shown as input and output of that algorithm. We let $y \leftarrow\!\$\, A(x_1,\ldots)$ denote picking r at random and letting $y \leftarrow A(x_1,\ldots;r)$. We omit the semicolon when there are no inputs other than the random coins. We let $[A(x_1,\ldots)]$ denote the set of all possible outputs of A when invoked with inputs $x_1,\ldots$. Adversaries are algorithms. The instruction $\mathbf{abort}(x_1,\ldots)$ is used to immediately halt with output $(x_1,\ldots)$.

We use a special symbol $\perp \notin \{0,1\}^*$ to denote an empty table position, and we also

$$
\begin{array}{l}
\underline{\text{Game } \mathrm{CR}_{\mathsf{H}}^{\mathcal{A}_{\mathsf{H}}}} \\
hk \leftarrow\$ \mathsf{H.Kg} \\
(m_0, m_1) \leftarrow\$ \mathcal{A}_{\mathsf{H}}(hk) \\
y_0 \leftarrow \mathsf{H.Ev}(hk, m_0) \\
y_1 \leftarrow \mathsf{H.Ev}(hk, m_1) \\
\text{Return } (m_0 \neq m_1) \text{ and } (y_0 = y_1)
\end{array}
$$

Figure 2.1. Game defining collision-resistance of function family H.

return it as an error code indicating an invalid input. An algorithm may not accept $\perp$ as input. If $x_i = \perp$ for some i when executing $(y_1, \dots) \leftarrow A(x_1 \dots)$ we assume that $y_j = \perp$ for all j. We assume that adversaries never pass $\perp$ as input to their oracles.

We use the code based game playing framework of [16]. (See Fig. 2.1 for an example of a game.) We let $\Pr[\mathrm{G}]$ denote the probability that game G returns true. In code, tables are initially empty. We adopt the convention that the running time of an adversary means the worst case execution time of the adversary in the game that executes it, so that time for game setup steps and time to compute answers to oracle queries is included.

Function families.

A family of functions H specifies algorithms $\mathsf{H.Kg}$ and $\mathsf{H.Ev}$, where $\mathsf{H.Ev}$ is deterministic. Key generation algorithm $\mathsf{H.Kg}$ returns a key hk, denoted by $hk \leftarrow\$ \mathsf{H.Kg}$. Evaluation algorithm $\mathsf{H.Ev}$ takes hk and an input $x \in \{0,1\}^*$ to return an output y, denoted by $y \leftarrow \mathsf{H.Ev}(hk, x)$.

Collision-resistant functions.

Consider game CR of Fig. 2.1 associated to a function family H and an adversary $\mathcal{A}_{\mathsf{H}}$. The game samples a random key hk for function family H. In order to win the game, adversary $\mathcal{A}_{\mathsf{H}}$ has to find two distinct messages m_0, m_1 such that $\mathsf{H.Ev}(hk, m_0) = \mathsf{H.Ev}(hk, m_1)$. The advantage of $\mathcal{A}_{\mathsf{H}}$ in breaking the CR security of H is defined as $\mathsf{Adv}_{\mathsf{H}}^{\mathsf{cr}}(\mathcal{A}_{\mathsf{H}}) = \Pr[\mathrm{CR}_{\mathsf{H}}^{\mathcal{A}_{\mathsf{H}}}]$.

Digital signature schemes.

A digital signature scheme DS specifies algorithms $\mathsf{DS.Kg}$, $\mathsf{DS.Sign}$ and $\mathsf{DS.Vrfy}$, where $\mathsf{DS.Vrfy}$ is deterministic. Associated to DS is a key generation randomness space $\mathsf{DS.KgRS}$

and signing algorithm's randomness space $\mathsf{DS.SignRS}$. Key generation algorithm $\mathsf{DS.Kg}$ takes randomness $z \in \mathsf{DS.KgRS}$ to return a signing key sk and a verification key vk, denoted by $(sk, vk) \leftarrow \mathsf{DS.Kg}(z)$. Signing algorithm $\mathsf{DS.Sign}$ takes sk, a message $m \in \{0,1\}^*$ and randomness $z \in \mathsf{DS.SignRS}$ to return a signature σ, denoted by $\sigma \leftarrow \mathsf{DS.Sign}(sk, m; z)$. Verification algorithm $\mathsf{DS.Vrfy}$ takes vk, σ, and m to return a decision $t \in \{\mathsf{true}, \mathsf{false}\}$ regarding whether σ is a valid signature of m under vk, denoted by $t \leftarrow \mathsf{DS.Vrfy}(vk, \sigma, m)$. The correctness condition for DS requires that $\mathsf{DS.Vrfy}(vk, \sigma, m) = \mathsf{true}$ for all $(sk, vk) \in [\mathsf{DS.Kg}]$, all $m \in \{0,1\}^*$, and all $\sigma \in [\mathsf{DS.Sign}(sk, m)]$.

We define the min-entropy of algorithm $\mathsf{DS.Kg}$ as $\mathrm{H}_\infty(\mathsf{DS.Kg})$, such that

$$2^{-\mathrm{H}_\infty(\mathsf{DS.Kg})} = \max_{vk \in \{0,1\}^*} \Pr\left[vk^* = vk : (sk^*, vk^*) \leftarrow\!\!\$\ \mathsf{DS.Kg}\right].$$

The probability is defined over the random coins used for $\mathsf{DS.Kg}$. Note that the min-entropy is defined with respect to verification keys, regardless of the corresponding values of the secret keys.

2.2 New asymmetric primitives

In this section we define key-updatable digital signatures and key-updatable public-key encryption. Both allow their keys to be updated with arbitrary strings. While in general one would prefer the size of keys, signatures, and ciphertexts to be constant, we will be willing to accept schemes for which these grow linearly in the number of updates. As we will discuss later, these are plausibly acceptable inefficiencies for our use cases.

We specify multi-user security definitions for both primitives, because it allows tighter reductions when we construct a channel from these primitives. Single-user variants of these definitions are obtained by only allowing the adversary to interact with one user and can be shown to imply the multi-user versions by a standard hybrid argument. Starting with [4] constructions have been given for a variety of primitives that allow multi-user security to be proven without

Game $\text{DSCORR}_{\mathsf{DS}}^{\mathcal{C}}$	Game $\text{PKECORR}_{\mathsf{PKE}}^{\mathcal{C}}$
$\vec{\Delta} \leftarrow ()$; $\nu \leftarrow\$ \mathsf{DS.KgRS}$	$\vec{\Delta} \leftarrow ()$; $\nu \leftarrow\$ \mathsf{PKE.KgRS}$
$(sk, vk) \leftarrow \mathsf{DS.Kg}(\nu)$	$(ek, dk) \leftarrow \mathsf{PKE.Kg}(\nu)$
$\mathcal{C}^{\text{UPD,SIGN}}(\nu)$	$\mathcal{C}^{\text{UPD,ENC}}(\nu)$
Return bad	Return bad
$\underline{\text{UPD}(\Delta)}$ / $\Delta \in \{0,1\}^*$	$\underline{\text{UPD}(\Delta)}$ / $\Delta \in \{0,1\}^*$
$\vec{\Delta} \leftarrow \vec{\Delta} \parallel \Delta$	$\vec{\Delta} \leftarrow \vec{\Delta} \parallel \Delta$
$sk \leftarrow\$ \mathsf{DS.UpdSk}(sk, \Delta)$	$dk \leftarrow\$ \mathsf{PKE.UpdDk}(dk, \Delta)$
Return sk	Return dk
$\underline{\text{SIGN}(m)}$ / $m \in \{0,1\}^*$	$\underline{\text{ENC}(m, \ell)}$ / $m, \ell \in \{0,1\}^*$
$\sigma \leftarrow\$ \mathsf{DS.Sign}(sk, m)$	$(ek^*, c) \leftarrow\$ \mathsf{PKE.Enc}(ek, \ell, m, \vec{\Delta})$
$(vk^*, t) \leftarrow \mathsf{DS.Vrfy}(vk, \sigma, m, \vec{\Delta})$	$m' \leftarrow \mathsf{PKE.Dec}(dk, \ell, c)$
If not t then bad $\leftarrow$ true	If $m' \neq m$ then bad $\leftarrow$ true

Figure 2.2. Games defining correctness of key-updatable digital signature scheme DS and correctness of key-updatable public-key encryption scheme PKE.

the factor q security loss introduced by a hybrid argument. If analogous constructions can be found for our primitives then our results will give tight bounds on the security of our channel.

2.2.1 Key-updatable digital signature schemes

We start by formally defining the syntax and correctness of a key-updatable digital signature scheme. Then we specify a security definition for it. We will briefly sketch how to construct such a scheme, but leave the details to Section 2.8.

Syntax and correctness.

A *key-updatable* digital signature scheme is a digital signature scheme with additional algorithms DS.UpdSk and DS.UpdVk, where DS.UpdVk is deterministic. Signing-key update algorithm DS.UpdSk takes a signing key sk and a key update information $\Delta \in \{0,1\}^*$ to return a new signing key sk, denoted by $sk \leftarrow\$ \mathsf{DS.UpdSk}(sk, \Delta)$. Verification-key update algorithm DS.UpdVk takes a verification key vk and a key update information $\Delta \in \{0,1\}^*$ to return a new verification key vk, denoted by $vk \leftarrow \mathsf{DS.UpdVk}(vk, \Delta)$.

Let $\vec{\Delta} = (\Delta_1, \Delta_2, \ldots, \Delta_n)$. For compactness, we sometimes use the notation $(vk, t) \leftarrow$

$\underline{\text{Game UNIQ}_{\mathsf{DS}}^{\mathcal{B}_{\mathsf{DS}}}}$
$(\Lambda, m, \sigma_1, \sigma_2, \vec{\Delta}) \leftarrow\!\!\$\, \mathcal{B}_{\mathsf{DS}}^{\text{NEWUSER}}$
$(sk, vk) \leftarrow \mathsf{DS.Kg}(z[\Lambda])$
$(vk^*, t_1) \leftarrow \mathsf{DS.Vrfy}(vk, \sigma_1, m, \vec{\Delta})$
$(vk^*, t_2) \leftarrow \mathsf{DS.Vrfy}(vk, \sigma_2, m, \vec{\Delta})$
Return $\sigma_1 \neq \sigma_2$ and t_1 and t_2

$\underline{\text{NEWUSER}(\Lambda)}$ // $\Lambda \in \{0,1\}^*$
If $z[\Lambda] \neq \perp$ then return $\perp$
$z[\Lambda] \leftarrow\!\!\$\, \mathsf{DS.KgRS}$
Return $z[\Lambda]$

Figure 2.3. Game defining signature uniqueness of key-updatable digital signature scheme DS.

$\mathsf{DS.Vrfy}(vk, \sigma, m, \vec{\Delta})$ to denote updating the verification key via $vk \leftarrow \mathsf{DS.UpdVk}(vk, \Delta_i)$ for $i = 1, \ldots, n$ and then evaluating $t \leftarrow \mathsf{DS.Vrfy}(vk, \sigma, m)$.

The key-update correctness condition requires that signatures must verify correctly as long as the signing and the verification keys are both updated with the same sequence of key update information $\vec{\Delta} = (\Delta_1, \Delta_2, \ldots)$. To formalize this, consider game DSCORR of Fig. 2.2, associated to a key-updatable digital signature scheme DS and an adversary $\mathcal{C}$. The advantage of an adversary $\mathcal{C}$ against the correctness of DS is given by $\mathsf{Adv}_{\mathsf{DS}}^{\mathsf{dscorr}}(\mathcal{C}) = \Pr[\text{DSCORR}_{\mathsf{DS}}^{\mathcal{C}}]$. We require that $\mathsf{Adv}_{\mathsf{DS}}^{\mathsf{dscorr}}(\mathcal{C}) = 0$ for all (even unbounded) adversaries $\mathcal{C}$. See Section 2.3 for discussion on game-based definitions of correctness.

Signature uniqueness.

We will be interested in schemes for which there is only a single signature that will be accepted for any message m and any sequence of updates $\vec{\Delta}$. Consider game UNIQ of Fig. 2.3, associated to a key-updatable digital signature scheme DS and an adversary $\mathcal{B}_{\mathsf{DS}}$. The adversary $\mathcal{B}_{\mathsf{DS}}$ can call the oracle NEWUSER arbitrarily many times with a user identifier Λ and be given the randomness used to generate the keys of Λ. The adversary ultimately outputs a user id Λ, a message m, signatures σ_1, σ_2, and a key update vector $\vec{\Delta}$. It wins if the signatures are distinct and both verify for m when the verification key of Λ is updated with $\vec{\Delta}$. The advantage of $\mathcal{B}_{\mathsf{DS}}$ in breaking the UNIQ security of DS is defined as $\mathsf{Adv}_{\mathsf{DS}}^{\mathsf{uniq}}(\mathcal{B}_{\mathsf{DS}}) = \Pr[\text{UNIQ}_{\mathsf{DS}}^{\mathcal{B}_{\mathsf{DS}}}]$.

Game $\text{UFEXP}_{\mathsf{DS}}^{\mathcal{A}_{\mathsf{DS}}}$	$\text{UPD}(\Lambda, \Delta)$ // $\Lambda, \Delta \in \{0,1\}^*$
$\mathsf{out} \leftarrow\!\$\, \mathcal{A}_{\mathsf{DS}}^{\text{NEWUSER,UPD,SIGN,EXP}}$ $(\Lambda, \sigma, m, \vec{\Delta}) \leftarrow \mathsf{out}$ $\mathsf{forgery} \leftarrow (\sigma, m, \vec{\Delta})$ $\mathsf{trivial} \leftarrow (\sigma^*[\Lambda], m^*[\Lambda], \vec{\Delta}^*[\Lambda])$ $t_1 \leftarrow (\mathsf{forgery} = \mathsf{trivial})$ $t_2 \leftarrow (\vec{\Delta}'[\Lambda] \sqsubseteq \vec{\Delta})$ $\mathsf{cheated} \leftarrow (t_1 \text{ or } t_2)$ $vk \leftarrow vk[\Lambda]$ $(vk, \mathsf{win}) \leftarrow \mathsf{DS.Vrfy}(vk, \sigma, m, \vec{\Delta})$ Return win and not cheated $\text{NEWUSER}(\Lambda)$ // $\Lambda \in \{0,1\}^*$ If $sk[\Lambda] \neq \perp$ then return $\perp$ $\vec{\Delta}_s[\Lambda] \leftarrow ()$; $\vec{\Delta}^*[\Lambda] \leftarrow \perp$; $\vec{\Delta}'[\Lambda] \leftarrow \perp$ $(sk[\Lambda], vk[\Lambda]) \leftarrow\!\$\, \mathsf{DS.Kg}$ Return $vk[\Lambda]$	If $sk[\Lambda] = \perp$ then return $\perp$ $\vec{\Delta}_s[\Lambda] \leftarrow \vec{\Delta}_s[\Lambda] \parallel \Delta$ $sk[\Lambda] \leftarrow\!\$\, \mathsf{DS.UpdSk}(sk[\Lambda], \Delta)$ Return $\perp$ $\text{SIGN}(\Lambda, m)$ // $\Lambda, m \in \{0,1\}^*$ If $sk[\Lambda] = \perp$ then return $\perp$ If $\vec{\Delta}^*[\Lambda] \neq \perp$ then return $\perp$ $\sigma \leftarrow\!\$\, \mathsf{DS.Sign}(sk[\Lambda], m)$ $(\sigma^*[\Lambda], m^*[\Lambda], \vec{\Delta}^*[\Lambda]) \leftarrow (\sigma, m, \vec{\Delta}_s[\Lambda])$ Return σ $\text{EXP}(\Lambda)$ // $\Lambda \in \{0,1\}^*$ If $sk[\Lambda] = \perp$ then return $\perp$ If $\vec{\Delta}'[\Lambda] = \perp$ then $\vec{\Delta}'[\Lambda] \leftarrow \vec{\Delta}_s[\Lambda]$ Return $sk[\Lambda]$

Game $\text{INDEXP}_{\mathsf{PKE}}^{\mathcal{A}_{\mathsf{PKE}}}$	$\text{ENC}(\Lambda, m_0, m_1, \ell)$ // $\Lambda, m_0, m_1, \ell \in \{0,1\}^*$
$b \leftarrow\!\$\, \{0,1\}$ $b' \leftarrow\!\$\, \mathcal{A}_{\mathsf{PKE}}^{\text{NEWUSER,UPDEK,UPDDK,ENC,DEC,EXP}}$ Return $b = b'$ $\text{NEWUSER}(\Lambda)$ // $\Lambda \in \{0,1\}^*$ If $dk[\Lambda] \neq \perp$ then return $\perp$ $\vec{\Delta}_e[\Lambda] \leftarrow ()$; $\vec{\Delta}_d[\Lambda] \leftarrow ()$ $\vec{\Delta}'[\Lambda] \leftarrow \perp$; $S[\Lambda] \leftarrow \emptyset$ $(ek[\Lambda], dk[\Lambda]) \leftarrow\!\$\, \mathsf{PKE.Kg}$ Return $ek[\Lambda]$ $\text{UPDEK}(\Lambda, \Delta)$ // $\Lambda, \Delta \in \{0,1\}^*$ If $dk[\Lambda] = \perp$ then return $\perp$ $ek[\Lambda] \leftarrow \mathsf{PKE.UpdEk}(ek[\Lambda], \Delta)$ $\vec{\Delta}_e[\Lambda] \leftarrow \vec{\Delta}_e[\Lambda] \parallel \Delta$ $\text{UPDDK}(\Lambda, \Delta)$ // $\Lambda, \Delta \in \{0,1\}^*$ If $dk[\Lambda] = \perp$ then return $\perp$ $dk[\Lambda] \leftarrow\!\$\, \mathsf{PKE.UpdDk}(dk[\Lambda], \Delta)$ $\vec{\Delta}_d[\Lambda] \leftarrow \vec{\Delta}_d[\Lambda] \parallel \Delta$	If $dk[\Lambda] = \perp$ then return $\perp$ If $\lvert m_0 \rvert \neq \lvert m_1 \rvert$ then return $\perp$ If $\vec{\Delta}'[\Lambda] \sqsubseteq \vec{\Delta}_e[\Lambda]$ then return $\perp$ $c \leftarrow\!\$\, \mathsf{PKE.Enc}(ek[\Lambda], \ell, m_b)$ $S[\Lambda] \leftarrow S[\Lambda] \cup \{(\vec{\Delta}_e[\Lambda], c, \ell)\}$ Return c $\text{DEC}(\Lambda, c, \ell)$ // $\Lambda, c, \ell \in \{0,1\}^*$ If $dk[\Lambda] = \perp$ then return $\perp$ If $(\vec{\Delta}_d[\Lambda], c, \ell) \in S[\Lambda]$ then return $\perp$ $m \leftarrow \mathsf{PKE.Dec}(dk[\Lambda], \ell, c)$ Return m $\text{EXP}(\Lambda)$ // $\Lambda \in \{0,1\}^*$ If $dk[\Lambda] = \perp$ then return $\perp$ If $\exists(\vec{\Delta}, c, \ell) \in S[\Lambda]$ s.t. $\vec{\Delta}_d[\Lambda] \sqsubseteq \vec{\Delta}$ then Return $\perp$ If $\vec{\Delta}'[\Lambda] = \perp$ then $\vec{\Delta}'[\Lambda] \leftarrow \vec{\Delta}_d[\Lambda]$ Return $dk[\Lambda]$

Figure 2.4. Games defining signature unforgeability under exposures of key-updatable digital signature scheme DS, and ciphertext indistinguishability under exposures of key-updatable public-key encryption scheme PKE.

Signature unforgeability under exposures.

Our main security notion for signatures asks that the adversary not be able to create signatures for any key update vector $\vec{\Delta}$ unless it acquired a signing key for some key update vector $\vec{\Delta}'$ that is a prefix of $\vec{\Delta}$. The adversary can query an oracle to receive a signature for one message with respect to each user's secret key, but it cannot claim this signature as a forgery. Consider game UFEXP of Fig. 2.4, associated to a key-updatable digital signature scheme DS and an adversary $\mathcal{A}_{\mathsf{DS}}$.

The adversary $\mathcal{A}_{\mathsf{DS}}$ can call the oracle NEWUSER arbitrarily many times, once for each user identifier Λ, and be given the verification key for that user. Then it can interact with user Λ via three different oracles. Via calls to UPD with a string Δ it requests that the signing key for the specified user be updated with Δ. Via calls to SIGN with message m it asks for a signature of m using the signing key for the specified user. When it does so, $\vec{\Delta}^*[\Lambda]$ is used to store the vector of strings the key was updated with, and no more signatures are allowed for user id Λ.[1] Via calls to EXP it can ask to be given the current signing key of the specified user. When it does so, $\vec{\Delta}'[\Lambda]$ is used to store the vector of strings the key was updated with.

At the end of the game the adversary outputs a user id Λ, a signature σ, a message m, and a key update vector $\vec{\Delta}$. The adversary has cheated if it previously received σ as the result of calling $\text{SIGN}(\Lambda, m)$ and $\vec{\Delta} = \vec{\Delta}^*[\Lambda]$, or if it exposed the signing key of Λ and $\vec{\Delta}'[\Lambda]$ is a prefix of $\vec{\Delta}$. It wins if it has not cheated and if σ verifies for m when the verification key of Λ is updated with $\vec{\Delta}$. The advantage of $\mathcal{A}_{\mathsf{DS}}$ in breaking the UFEXP security of DS is defined by $\mathsf{Adv}^{\mathsf{ufexp}}_{\mathsf{DS}}(\mathcal{A}_{\mathsf{DS}}) = \Pr[\text{UFEXP}^{\mathcal{A}_{\mathsf{DS}}}_{\mathsf{DS}}]$.

Construction.

In Section 2.8 we use a forward-secure [9] key-evolving signature scheme with unique signatures to construct a key-updatable signature scheme that is secure with respect to both of

[1] We are thus defining security for a *one-time* signature scheme, because a particular key will only be used for one signature. This is all we require for our application, but the definition and construction we provide could easily be extended to allow multiple signatures if desired.

the above definitions. Roughly, a key-evolving signature scheme is like a key-updatable digital signature scheme that can only update with $\Delta = \varepsilon$. In order to enable updates with respect to arbitrary key update information, we sign each update string with the current key prior to evolving the key, and then include these intermediate signatures with our final signature.

2.2.2 Key-updatable public-key encryption schemes

We start by formally defining the syntax and correctness of a key-updatable public-key encryption. Then we specify a security definition for it. We will briefly sketch how to construct such a scheme, but leave the details to Section 2.9. We consider public-key encryption with labels as introduced by Shoup [62].

Syntax and correctness.

A key-updatable public-key encryption scheme PKE specifies algorithms $\mathsf{PKE.Kg}$, $\mathsf{PKE.Enc}$, $\mathsf{PKE.Dec}$, $\mathsf{PKE.UpdEk}$, and $\mathsf{PKE.UpdDk}$. Associated to PKE is a key generation randomness space $\mathsf{PKE.KgRS}$, encryption randomness space $\mathsf{PKE.EncRS}$, and decryption-key update randomness space $\mathsf{PKE.UpdDkRS}$. Key generation algorithm $\mathsf{PKE.Kg}$ takes randomness $z \in \mathsf{PKE.KgRS}$ to return an encryption key ek and a decryption key dk, denoted by $(ek, dk) \leftarrow \mathsf{PKE.Kg}(z)$. Encryption algorithm $\mathsf{PKE.Enc}$ takes ek, a label $\ell \in \{0,1\}^*$, a message $m \in \{0,1\}^*$, and randomness $z \in \mathsf{PKE.EncRS}$ to return a ciphertext c, denoted by $c \leftarrow \mathsf{PKE.Enc}(ek, \ell, m; z)$. Deterministic decryption algorithm $\mathsf{PKE.Dec}$ takes dk, ℓ, c to return a message $m \in \{0,1\}^*$, denoted by $m \leftarrow \mathsf{PKE.Dec}(dk, \ell, c)$. Deterministic encryption-key update algorithm $\mathsf{PKE.UpdEk}$ takes an encryption key ek and key update information $\Delta \in \{0,1\}^*$ to return a new encryption key ek, denoted by $ek \leftarrow \mathsf{PKE.UpdEk}(ek, \Delta)$. Decryption-key update algorithm $\mathsf{PKE.UpdDk}$ takes a decryption key dk, key update information $\Delta \in \{0,1\}^*$, and randomness $z \in \mathsf{PKE.UpdDkRS}$ to return a new decryption key dk, denoted by $dk \leftarrow \mathsf{PKE.UpdDk}(dk, \Delta; z)$.

Let $\vec{\Delta} = (\Delta_1, \Delta_2, \ldots, \Delta_n)$. For compactness, we sometimes use the notation $(ek, c) \leftarrow\$$ $\mathsf{PKE.Enc}(ek, \ell, m, \vec{\Delta})$ to denote updating the key via $ek \leftarrow \mathsf{PKE.UpdEk}(ek, \Delta_i)$ for $i = 1, \ldots, n$

and then evaluating $c \leftarrow\!\!\$\ \mathsf{PKE.Enc}(ek, \ell, m)$.

The correctness condition requires that ciphertexts decrypt correctly as long as the encryption and decryption key are both updated with the same sequence of key update information $\vec{\Delta} = (\Delta_1, \Delta_2, \ldots)$. To formalize this, consider game PKECORR of Fig. 2.2, associated to a key-updatable public-key encryption scheme PKE and an adversary $\mathcal{C}$. The advantage of an adversary $\mathcal{C}$ against the correctness of PKE is given by $\mathsf{Adv}^{\mathsf{pkecorr}}_{\mathsf{PKE}}(\mathcal{C}) = \Pr[\mathrm{PKECORR}^{\mathcal{C}}_{\mathsf{PKE}}]$. Correctness requires that $\mathsf{Adv}^{\mathsf{pkecorr}}_{\mathsf{PKE}}(\mathcal{C}) = 0$ for all (even computationally unbounded) adversaries $\mathcal{C}$. See Section 2.3 for discussion on game-based definitions of correctness.

We denote the min-entropy of algorithms PKE.Kg and PKE.Enc as $\mathrm{H}_\infty(\mathsf{PKE.Kg})$ and $\mathrm{H}_\infty(\mathsf{PKE.Enc})$, respectively, which are defined as follows:

$$2^{-\mathrm{H}_\infty(\mathsf{PKE.Kg})} = \max_{ek} \Pr\left[ek^* = ek : (ek^*, dk^*) \leftarrow\!\!\$\ \mathsf{PKE.Kg}\right],$$

$$2^{-\mathrm{H}_\infty(\mathsf{PKE.Enc})} = \max_{ek,\ell,m,c} \Pr\left[c^* = c : c^* \leftarrow\!\!\$\ \mathsf{PKE.Enc}(ek, \ell, m)\right].$$

The probability is defined over the random coins used by PKE.Kg and PKE.Enc, respectively. Note that min-entropy of PKE.Kg does not depend on the output value dk^*.

Ciphertext indistinguishability under exposures.

Consider game INDEXP of Fig. 2.4, associated to a key-updatable public-key encryption scheme PKE and an adversary $\mathcal{A}_{\mathsf{PKE}}$. Roughly, it requires that PKE maintain CCA security [6] even if $\mathcal{A}_{\mathsf{PKE}}$ is given the decryption key (as long as that decryption key is no longer able to decrypt any challenge ciphertexts).

The adversary $\mathcal{A}_{\mathsf{PKE}}$ can call the oracle NEWUSER arbitrarily many times with a user identifier Λ and be given the encryption key of that user. Then it can interact with user Λ via five oracles. Via calls to UPDEK or UPDDK with key update information Δ it requests for the corresponding key to be updated with that string. Variable $\vec{\Delta}_e$ stores the sequence of update information used to update the encryption key and $\vec{\Delta}_d$ the sequence of update information used

to update the decryption key.

Via calls to $\textsc{Enc}$ with messages m_0, m_1 and label ℓ it requests that one of these messages be encrypted using the specified label (which message is encrypted depends on the secret bit b). It will be given back the produced ciphertext. Set S is used to store the challenge ciphertext, label, and current value of $\vec{\Delta}_e$.

Via calls to $\textsc{Dec}$ with ciphertext c and ℓ it requests that the ciphertext be decrypted with the specified label. Adversary $\mathcal{A}_{\mathsf{PKE}}$ is not allowed to make such a query with a pair (c, ℓ) obtained from a call to $\textsc{Enc}$ if $\vec{\Delta}_d$ is the same as $\vec{\Delta}_e$ was at the time of encryption.

Via calls to $\textsc{Exp}$ it asks to be given the current decryption key of the user. It may not do so if $\vec{\Delta}_d$ is a prefix of any $\vec{\Delta}_e$ when a $\textsc{Enc}$ query was made. Variable $\vec{\Delta}'$ is used to disallow future calls to $\textsc{Enc}$ of this form.

At the end of the game the adversary outputs a bit b' representing its guess of the secret bit b. The advantage of $\mathcal{A}_{\mathsf{PKE}}$ in breaking the INDEXP security of PKE is defined as $\mathsf{Adv}^{\mathsf{indexp}}_{\mathsf{PKE}}(\mathcal{A}_{\mathsf{PKE}}) = 2\Pr[\mathrm{INDEXP}^{\mathcal{A}_{\mathsf{PKE}}}_{\mathsf{PKE}}] - 1$.

Construction.

In Section 2.9 we use a hierarchical identity-based encryption (HIBE) scheme to construct a secure key-updatable encryption scheme. Roughly, a HIBE assigns a decryption key to any identity (vector of strings). A decryption key for an identity $\vec{I}$ can be used to create decryption keys for an identity of which $\vec{I}$ is a prefix. Security requires that the adversary be unable to learn about encrypted messages encrypted to an identity $\vec{I}$ even if given the decryption key for many identities as long as none of them were prefixes of $\vec{I}$. To create a key-updatable encryption scheme we treat the vector of key updates as an identity. The security of this scheme then follows from the security of the underlying HIBE in a fairly straightforward manner.

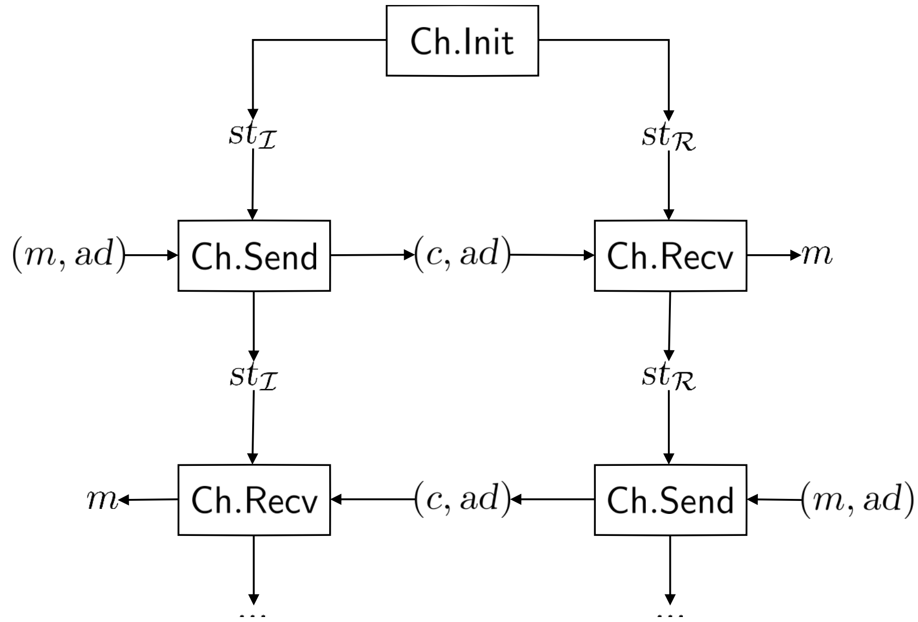

Figure 2.5. A basic interaction between bidirectional cryptographic channel algorithms.

2.3 Bidirectional cryptographic channels

In this section we formally define the syntax and correctness of bidirectional cryptographic channels. Our notion of bidirectional channels will closely match that of Marson and Poettering [49]. Compared to their definition, we allow the receiving algorithm to be randomized and provide an alternative correctness condition. We argue that the new correctness condition is more appropriate for our desired use case of secure messaging. Henceforth, we will omit the adjective "bidirectional" and refer simply to channels.

Syntax of channel.

A channel provides a method for two users to exchange messages in an arbitrary order. We will refer to the two users of a channel as the initiator $\mathcal{I}$ and the receiver $\mathcal{R}$. There will be no formal distinction between the two users, but when specifying attacks we follow the convention of having $\mathcal{I}$ send a ciphertext first. We will use u as a variable to represent an arbitrary user and $\overline{u}$ to represent the other user. More formally, when $u \in \{\mathcal{I}, \mathcal{R}\}$ we let $\overline{u}$ denote the sole element of $\{\mathcal{I}, \mathcal{R}\} \setminus \{u\}$. Consider Fig. 2.5 for an overview of algorithms that constitute a channel Ch,

and the interaction between them.

A channel Ch specifies algorithms $\mathsf{Ch.Init}$, $\mathsf{Ch.Send}$, and $\mathsf{Ch.Recv}$. Initialization algorithm $\mathsf{Ch.Init}$ returns initial states $st_{\mathcal{I}} \in \{0,1\}^*$ and $st_{\mathcal{R}} \in \{0,1\}^*$, where $st_{\mathcal{I}}$ is $\mathcal{I}$'s state and $st_{\mathcal{R}}$ is $\mathcal{R}$'s state. We write $(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\$ \mathsf{Ch.Init}$. Sending algorithm $\mathsf{Ch.Send}$ takes state $st_u \in \{0,1\}^*$, associated data $ad \in \{0,1\}^*$, and message $m \in \{0,1\}^*$ to return updated state $st_u \in \{0,1\}^*$ and a ciphertext $c \in \{0,1\}^*$. We write $(st_u, c) \leftarrow\$ \mathsf{Ch.Send}(st_u, ad, m)$. Receiving algorithm takes state $st_u \in \{0,1\}^*$, associated data $ad \in \{0,1\}^*$, and ciphertext $c \in \{0,1\}^*$ to return updated state $st_u \in \{0,1\}^* \cup \{\bot\}$ and message $m \in \{0,1\}^* \cup \{\bot\}$. We write $(st_u, m) \leftarrow\$ \mathsf{Ch.Recv}(st_u, ad, c)$, where $m = \bot$ represents a rejection of ciphertext c and $st_u = \bot$ represents the channel being permanently shut down from the perspective of u (recall our convention regarding $\bot$ as input to an algorithm). One notion of correctness we discuss will require that $st_u = \bot$ whenever $m = \bot$. The other will require that st_u not be changed from its input value when $m = \bot$.

We let $\mathsf{Ch.InitRS}$, $\mathsf{Ch.SendRS}$, and $\mathsf{Ch.RecvRS}$ denote the sets of possible random coins for $\mathsf{Ch.Init}$, $\mathsf{Ch.Send}$, and $\mathsf{Ch.Recv}$, respectively. Note that for full generality we allow $\mathsf{Ch.Recv}$ to be randomized. Prior work commonly requires this algorithm to be deterministic.

Correctness of channel.

In Fig. 2.6 we provide two games, defining two alternative correctness requirements for a cryptographic channel. Lines labelled with the name of a game are included only in that game. The games differ in whether the adversary is given access to an oracle $\textsc{Robust}$ or to an oracle $\textsc{Reject}$. Game CORR uses the former, whereas game CORR⊥ uses the latter. The advantage of an adversary $\mathcal{C}$ against the correctness of channel Ch is given by $\mathsf{Adv}^{\mathsf{corr}}_{\mathsf{Ch}}(\mathcal{C}) = \Pr[\mathrm{CORR}^{\mathcal{C}}_{\mathsf{Ch}}]$ in one case, and $\mathsf{Adv}^{\mathsf{corr}\bot}_{\mathsf{Ch}}(\mathcal{C}) = \Pr[\mathrm{CORR}\bot^{\mathcal{C}}_{\mathsf{Ch}}]$ in the other case. Correctness with respect to either notion requires that the advantage is equal 0 for all (even computationally unbounded) adversaries $\mathcal{C}$.

Our use of games to define correctness conditions follows the work of Marson and Poettering [49] and Bellare et. al. [17]. By considering unbounded adversaries and requiring

$\underline{\text{Games } \mathrm{CORR}^{\mathcal{C}}_{\mathsf{Ch}}, \mathrm{CORR}\bot^{\mathcal{C}}_{\mathsf{Ch}}}$
$s_{\mathcal{I}} \leftarrow r_{\mathcal{I}} \leftarrow s_{\mathcal{R}} \leftarrow r_{\mathcal{R}} \leftarrow 0\,;\ \nu \leftarrow\!\!{\scriptstyle\$}\ \mathsf{Ch.InitRS}$
$(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow \mathsf{Ch.Init}(\nu)$
$\mathcal{C}^{\textsc{Send},\textsc{Recv},\textsc{Robust}}(\nu)$ **/** $\mathrm{CORR}^{\mathcal{C}}_{\mathsf{Ch}}$
$\mathcal{C}^{\textsc{Send},\textsc{Recv},\textsc{Reject}}(\nu)$ **/** $\mathrm{CORR}\bot^{\mathcal{C}}_{\mathsf{Ch}}$
Return bad

$\underline{\textsc{Send}(u, ad, m)}$ **/** $u \in \{\mathcal{I}, \mathcal{R}\}, (ad, m) \in (\{0,1\}^*)^2$
$s_u \leftarrow s_u + 1\,;\ z \leftarrow\!\!{\scriptstyle\$}\ \mathsf{Ch.SendRS}\,;\ (st_u, c) \leftarrow \mathsf{Ch.Send}(st_u, ad, m; z)$
$\mathbf{ctable}_{\overline{u}}[s_u] \leftarrow (c, ad)\,;\ \mathbf{mtable}_{\overline{u}}[s_u] \leftarrow m\,;$ Return z

$\underline{\textsc{Recv}(u)}$ **/** $u \in \{\mathcal{I}, \mathcal{R}\}$
If $\mathbf{ctable}_u[r_u + 1] = \bot$ then return $\bot$
$r_u \leftarrow r_u + 1\,;\ (c, ad) \leftarrow \mathbf{ctable}_u[r_u]$
$\eta \leftarrow\!\!{\scriptstyle\$}\ \mathsf{Ch.RecvRS}\,;\ (st_u, m) \leftarrow \mathsf{Ch.Recv}(st_u, ad, c; \eta)$
If $m \neq \mathbf{mtable}_u[r_u]$ then bad $\leftarrow$ true
Return η

$\underline{\textsc{Robust}(\vec{k}, ad, c)}$ **/** $(\vec{k}, ad, c) \in (\{0,1\}^*)^3$
$(\vec{k}', m) \leftarrow\!\!{\scriptstyle\$}\ \mathsf{Ch.Recv}(\vec{k}, ad, c)$
If $m = \bot$ and $\vec{k}' \neq \vec{k}$ then bad $\leftarrow$ true

$\underline{\textsc{Reject}(\vec{k}, ad, c)}$ **/** $(\vec{k}, ad, c) \in (\{0,1\}^*)^3$
$(\vec{k}', m) \leftarrow\!\!{\scriptstyle\$}\ \mathsf{Ch.Recv}(\vec{k}, ad, c)$
If $m = \bot$ and $\vec{k}' \neq \bot$ then bad $\leftarrow$ true

Figure 2.6. Games defining correctness of channel Ch. Lines labelled with the name of a game are included only in that game.

---

an advantage of 0 we capture a typical information-theoretic perfect correctness requirement without having to explicitly quantify over sequences of actions. In this work we require only the perfect correctness because it is achieved by our scheme; however, it would be possible to capture computational correctness by considering a restricted class of adversaries.

Both games require that ciphertexts sent by any user are always decrypted to the correct message by the other user. This is modeled by providing adversary $\mathcal{C}$ with access to oracles $\textsc{Send}$ and $\textsc{Recv}$. We assume that messages from u to $\overline{u}$ are received in the same order they were sent, and likewise that messages from $\overline{u}$ to u are also received in the correct order (regardless Aof how they are interwoven on both sides, since ciphertexts are being sent in both directions).

The games differ in how the channel is required to behave in the case that a ciphertext is

rejected. Game CORR (using oracle ROBUST) requires that the state of the user not be changed so that the channel can continue to be used. Game CORR$\bot$ (using oracle REJECT) requires that the state of the user is set to $\bot$. According to our conventions about the behavior of algorithms given $\bot$ as input (see Section 2.1), the channel will then refuse to perform any further actions by setting all subsequent outputs to $\bot$. We emphasize that the adversary specifies all inputs to Ch.Recv when making calls to ROBUST and REJECT, so the behavior of those oracles is not related to the behavior of the other two oracles for which the game maintains the state of both users.

Comparison of correctness notions.

The correctness required by CORR$\bot$ is identical to that of Marson and Poettering [49]. The CORR notion of correctness instead uses a form of robustness analogous to that of [17]. In Section 2.7 we discuss how these correctness notions have different implications for the *security* of the channel. It is trivial to convert a CORR-correct channel to a CORR$\bot$-correct channel and vice versa. Thus we will, without loss of generality, only provide a scheme achieving CORR-correctness.

2.4 Security notion for channels

In this section we will define what it means for a channel to be secure in the presence of a strong attacker that can steal the secrets of either party in the communication. Our goal is to give the strongest possible notion of security in this setting, encompassing both the privacy of messages and the integrity of ciphertexts. We take a fine-grained look at what attacks are possible and require that a channel be secure against all attacks that are not syntactically inherent in the definition of a channel.

To introduce our security notion we will first describe a simple interface of how the adversary is allowed to interact with the channel. Then we show attacks that would break the security of *any* channel using this interface. Our final security notion will be created by adding

Game $\text{INTER}^{\mathcal{D}}_{\mathsf{Ch}}$	$\text{RECV}(u, c, ad)$

$b \leftarrow\!\!\$ \{0,1\}$
$s_{\mathcal{I}} \leftarrow r_{\mathcal{I}} \leftarrow s_{\mathcal{R}} \leftarrow r_{\mathcal{R}} \leftarrow 0$
$(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\!\!\$ \mathsf{Ch.Init}$
$(z_{\mathcal{I}}, z_{\mathcal{R}}) \leftarrow\!\!\$ (\mathsf{Ch.SendRS})^2$
$(\eta_{\mathcal{I}}, \eta_{\mathcal{R}}) \leftarrow\!\!\$ (\mathsf{Ch.RecvRS})^2$
$b' \leftarrow\!\!\$ \mathcal{D}^{\text{SEND},\text{RECV},\text{EXP}}$
Return $(b' = b)$
 	
$\text{SEND}(u, m_0, m_1, ad)$
/ $u \in \{\mathcal{I}, \mathcal{R}\}, (m_0, m_1, ad) \in (\{0,1\}^*)^3$
If $\mathsf{nextop} \neq (u, \text{“send”})$
and $\mathsf{nextop} \neq \bot$ then return $\bot$
If $\lvert m_0 \rvert \neq \lvert m_1 \rvert$ then return $\bot$
$(st_u, c) \leftarrow \mathsf{Ch.Send}(st_u, ad, m_b; z_u)$
$\mathsf{nextop} \leftarrow \bot$
$s_u \leftarrow s_u + 1$; $z_u \leftarrow\!\!\$ \mathsf{Ch.SendRS}$
$\mathbf{ctable}_{\overline{u}}[s_u] \leftarrow (c, ad)$
Return c
	/ $u \in \{\mathcal{I}, \mathcal{R}\}, (c, ad) \in (\{0,1\}^*)^2$
	If $\mathsf{nextop} \neq (u, \text{“recv”})$
	and $\mathsf{nextop} \neq \bot$ then return $\bot$
	$(st_u, m) \leftarrow \mathsf{Ch.Recv}(st_u, ad, c; \eta_u)$
	$\mathsf{nextop} \leftarrow \bot$; $\eta_u \leftarrow\!\!\$ \mathsf{Ch.RecvRS}$
	If $m \neq \bot$ then $r_u \leftarrow r_u + 1$
	If $b = 0$ and $(c, ad) \neq \mathbf{ctable}_u[r_u]$ then
	Return m
	Return $\bot$
	
	$\text{EXP}(u, \mathsf{rand})$
	/ $u \in \{\mathcal{I}, \mathcal{R}\}, \mathsf{rand} \in \{\varepsilon, \text{“send”}, \text{“recv”}\}$
	If $\mathsf{nextop} \neq \bot$ then return $\bot$
	$(z, \eta) \leftarrow (\varepsilon, \varepsilon)$
	If $\mathsf{rand} = \text{“send”}$ then
	$\mathsf{nextop} \leftarrow (u, \text{“send”})$; $z \leftarrow z_u$
	Else if $\mathsf{rand} = \text{“recv”}$ then
	$\mathsf{nextop} \leftarrow (u, \text{“recv”})$; $\eta \leftarrow \eta_u$
	Return (st_u, z, η)

Figure 2.7. Game defining interface between adversary $\mathcal{D}$ and channel Ch.

checks to the interface that prevents adversary from performing any sequence of actions that leads to these unpreventable breaches of security. We introduce only the minimal necessary restrictions preventing the attacks, making sure that we allow *all* adversaries that do not trivially break the security as per above.

2.4.1 Channel interface game

Consider game INTER in Fig. 2.7. It defines the interface between an adversary $\mathcal{D}$ and a channel Ch. A secret bit b is chosen at random and the adversary's goal is to guess this bit given access to a left-or-right sending oracle, real-or-$\bot$ receiving oracle, and an exposure oracle. The sending oracle takes as input a user $u \in \{\mathcal{I}, \mathcal{R}\}$, two messages $m_0, m_1 \in \{0,1\}^*$, and associated data ad. Then it returns the encryption of m_b with ad by user u. The receiving oracle RECV takes as input a user u, a ciphertext c, and associated data ad. It has user u decrypt this ciphertext using ad, and proceeds as follows. If $b = 0$ holds (along with another condition we discuss

momentarily) then it returns the valid decryption of this ciphertext; otherwise it returns $\perp$. The exposure oracle $\textsc{Exp}$ takes as input a user u, and a flag rand. It returns user's state st_u, and it might return random coins that will be used the next time this user runs algorithms $\mathsf{Ch.Send}$ or $\mathsf{Ch.Recv}$ (depending on the value of rand, which we discuss below). The advantage of adversary $\mathcal{D}$ against channel Ch is defined by $\mathsf{Adv}^{\mathsf{inter}}_{\mathsf{Ch}}(\mathcal{D}) = 2\Pr[\mathrm{INTER}^{\mathcal{D}}_{\mathsf{Ch}}] - 1$.

This interface gives the adversary full control over the communication between the two users of the channel. It may modify, reorder, or block any communication as it sees fit. The adversary is able to exfiltrate the secret state of either party at any time.

Let us consider the different cases of how a user's secrets might be exposed. They could be exposed while the user is in the middle of performing a $\mathsf{Ch.Send}$ operation, in the middle of performing a $\mathsf{Ch.Recv}$ operation, or when the user is idle (i.e. not in the middle of performing $\mathsf{Ch.Send}$ or $\mathsf{Ch.Recv}$). In the last case we expect the adversary to learn the user's state st_u, but nothing else. If the adversary is exposing the user during an operation, they would potentially learn the state before the operation, any secrets computed during the operation, and the state after the operation. We capture this by leaking the state from before the operation along with the randomness that will be used when the adversary makes its next query to $\textsc{Send}$ or $\textsc{Recv}$. This allows the adversary to compute the next state as well. The three possible values of rand are $\mathsf{rand} =$ "send" for the first possibility, $\mathsf{rand} =$ "recv" for the second possibility, and $\mathsf{rand} = \varepsilon$ for the third. These exposures represent what the adversary is learning while a particular operation is occurring, so we require (via nextop) that after such an exposure it immediately makes the corresponding oracle query. Without the use of the exposure oracle the game specified by this interface would essentially be equivalent to the combination of the integrity and confidentiality security notions defined by Marson and Poettering [49] in the all-in-one definition style of Rogaway and Shrimpton [59].

The interface game already includes some standard checks. First, we require that on any query (u, m_0, m_1, ad) to $\textsc{Send}$ the adversary must provide equal length messages. If the adversary does not do so (i.e. $|m_0| \neq |m_1|$) then $\textsc{Send}$ returns $\perp$ immediately. This prevents

the inherent attack where an adversary could distinguish between the two values of *b* by asking for encryptions of different length messages and checking the length of the output ciphertext. Adversary $\mathcal{D}_1$ in Fig. 2.8 does just that and would achieve $\mathsf{Adv}^{\mathsf{inter}}_{\mathsf{Ch}}(\mathcal{D}_1) > 1/2$ against any channel Ch if not for that check.

Second, we want to prevent $\textsc{Recv}$ from decrypting ciphertexts that are simply forwarded to it from $\textsc{Send}$. So for each user u we keep track of counters s_u and r_u that track how many messages that user has sent and received. Then at the end of a $\textsc{Send}$ call to u the ciphertext-associated data pair (c, ad) is stored in the table $\mathbf{ctable}_{\overline{u}}$ with index s_u. When $\textsc{Recv}$ is called for user $\overline{u}$ it will compare the pair (c, ad) against $\mathbf{ctable}_{\overline{u}}[r_{\overline{u}}]$ and if the pair matches return $\perp$ regardless of the value of the secret bit. If we did not do this check then for any channel Ch the adversary $\mathcal{D}_2$ shown in Fig. 2.8 would achieve $\mathsf{Adv}^{\mathsf{inter}}_{\mathsf{Ch}}(\mathcal{D}_2) = 1$.

We now specify several efficient adversaries that will have high advantage for *any* choice of Ch. For concreteness we always have our adversaries immediately start the actions required to perform the attacks, but all of the attacks would still work if the adversary had performed a number of unrelated procedure calls first. Associated data will never be important for our attacks so we will always set it to ε. We will typically set $m_0 = 0$ and $m_1 = 1$. For the following we let Ch be any channel and consider the adversaries shown in Fig. 2.8.

Trivial Forgery.

If the adversary exposes the secrets of u it will be able to forge a ciphertext that $\overline{u}$ would accept at least until the future point in time when $\overline{u}$ has received the ciphertext that u creates next. For a simple example of this consider the third adversary, $\mathcal{D}_3$. It exposes the secrets of user $\mathcal{I}$, then uses them to perform its own $\mathsf{Ch.Send}$ computation locally, and sends the resulting ciphertext to $\mathcal{R}$. Clearly this ciphertext will always decrypt to a non-$\perp$ value so the adversary can trivially determine the value of *b* and achieve $\mathsf{Adv}^{\mathsf{inter}}_{\mathsf{Ch}}(\mathcal{D}_3) = 1$.

After an adversary has done the above to trivially send a forgery to $\overline{u}$ it can easily perform further attacks on both the integrity and authenticity of the channel. These are shown

Adversary $\mathcal{D}_1^{\textsc{Send},\textsc{Recv},\textsc{Exp}}$

$(\vec{k}, z, \eta) \leftarrow \textsc{Exp}(\mathcal{I}, \varepsilon)$
$n \leftarrow \max_{c \in [\mathsf{Ch.Send}(\vec{k},\varepsilon,1)]} |c|$
$m \leftarrow\!\!\$\ \{0,1\}^{n+2}$
$c \leftarrow \textsc{Send}(\mathcal{I}, m, 1, \varepsilon)$
If $|c| \leq n$ then return 1
Return 0

Adversary $\mathcal{D}_2^{\textsc{Send},\textsc{Recv},\textsc{Exp}}$

$c \leftarrow \textsc{Send}(\mathcal{I}, 1, 1, \varepsilon)$
$m \leftarrow \textsc{Recv}(\mathcal{R}, c, \varepsilon)$
If $m = \perp$ then return 1
Return 0

Adversary $\mathcal{D}_3^{\textsc{Send},\textsc{Recv},\textsc{Exp}}$

$(\vec{k}, z, \eta) \leftarrow \textsc{Exp}(\mathcal{I}, \varepsilon)$
$(\vec{k}, c) \leftarrow\!\!\$\ \mathsf{Ch.Send}(\vec{k}, \varepsilon, 1)$
$m \leftarrow \textsc{Recv}(\mathcal{R}, c, \varepsilon)$
If $m = \perp$ then return 1
Return 0

Adversary $\mathcal{D}_{3.1}^{\textsc{Send},\textsc{Recv},\textsc{Exp}}$

$(\vec{k}, z, \eta) \leftarrow \textsc{Exp}(\mathcal{I}, \varepsilon)$
$(\vec{k}, c) \leftarrow\!\!\$\ \mathsf{Ch.Send}(\vec{k}, \varepsilon, 1)$
$m \leftarrow \textsc{Recv}(\mathcal{R}, c, \varepsilon)$
$(\vec{k}, c) \leftarrow\!\!\$\ \mathsf{Ch.Send}(\vec{k}, \varepsilon, 1)$
$m \leftarrow \textsc{Recv}(\mathcal{R}, c, \varepsilon)$
If $m = \perp$ then return 1
Return 0

Adversary $\mathcal{D}_{3.2}^{\textsc{Send},\textsc{Recv},\textsc{Exp}}$

$(\vec{k}, z, \eta) \leftarrow \textsc{Exp}(\mathcal{I}, \varepsilon)$
$(\vec{k}, c) \leftarrow\!\!\$\ \mathsf{Ch.Send}(\vec{k}, \varepsilon, 1)$
$m \leftarrow \textsc{Recv}(\mathcal{R}, c, \varepsilon)$
$c \leftarrow \textsc{Send}(\mathcal{R}, 0, 1, \varepsilon)$
$(\vec{k}, m) \leftarrow\!\!\$\ \mathsf{Ch.Recv}(\vec{k}, \varepsilon, c)$
If $m = 1$ then return 1
Return 0

Adversary $\mathcal{D}_4^{\textsc{Send},\textsc{Recv},\textsc{Exp}}$

$c \leftarrow \textsc{Send}(\mathcal{I}, 0, 1, \varepsilon)$
$(\vec{k}, z, \eta) \leftarrow \textsc{Exp}(\mathcal{R}, \varepsilon)$
$(\vec{k}, m) \leftarrow\!\!\$\ \mathsf{Ch.Recv}(\vec{k}, \varepsilon, c)$
If $m = 1$ then return 1
Return 0

Adversary $\mathcal{D}_5^{\textsc{Send},\textsc{Recv},\textsc{Exp}}$

$(\vec{k}, z, \eta) \leftarrow \textsc{Exp}(\mathcal{R}, \varepsilon)$
$c \leftarrow \textsc{Send}(\mathcal{I}, 0, 1, \varepsilon)$
$(\vec{k}, m) \leftarrow\!\!\$\ \mathsf{Ch.Recv}(\vec{k}, \varepsilon, c)$
If $m = 1$ then return 1
Return 0

Adversary $\mathcal{D}_6^{\textsc{Send},\textsc{Recv},\textsc{Exp}}$

$(\vec{k}, z, \eta) \leftarrow \textsc{Exp}(\mathcal{I}, \text{"send"})$
$(\vec{k}, c) \leftarrow \mathsf{Ch.Send}(\vec{k}, \varepsilon, 1; z)$
$c' \leftarrow \textsc{Send}(\mathcal{I}, 0, 1, \varepsilon)$
If $c' = c$ then return 1
Return 0

Figure 2.8. Generic attacks against any channel Ch with interface INTER.

by adversaries $\mathcal{D}_{3.1}$ and $\mathcal{D}_{3.2}$. The first displays the fact that the attacker can easily send further forgeries to $\overline{u}$. The second displays the fact that the attacker can now easily decrypt any messages sent by $\overline{u}$. We have $\mathsf{Adv}^{\mathsf{inter}}_{\mathsf{Ch}}(\mathcal{D}_{3.1}) = 1$ and $\mathsf{Adv}^{\mathsf{inter}}_{\mathsf{Ch}}(\mathcal{D}_{3.2}) = 1$.

Trivial Challenges.

If the adversary exposes the secrets of u it will necessarily be able to decrypt any ciphertexts already encrypted by $\overline{u}$ that have not already been received by u. Consider the adversary $\mathcal{D}_4$. It determines what message was encrypted by user $\mathcal{I}$ by exposing the state of $\mathcal{R}$, and uses that to run Ch.Recv. We have $\mathsf{Adv}^{\mathsf{inter}}_{\mathsf{Ch}}(\mathcal{D}_4) = 1$.

Similarly, if the adversary exposes the secrets of u it will necessarily be able to decrypt any future ciphertexts encrypted by $\overline{u}$, until $\overline{u}$ receives the ciphertext that u creates next. Consider the adversary $\mathcal{D}_5$. It is essentially the identical to adversary $\mathcal{D}_4$, except it reverses the order of the calls made to $\textsc{Send}$ and $\textsc{Exp}$. We have $\mathsf{Adv}^{\mathsf{inter}}_{\mathsf{Ch}}(\mathcal{D}_5) = 1$.

Exposing Randomness.

If an adversary exposes user u with $\mathsf{rand} =$ "send" then it is able to compute the next state of u by running $\mathsf{Ch.Send}$ locally with the same randomness that u will use. So in this case the security game must act as if the adversary exposed both the current and the next state. In particular, the attacks above could only succeed until, first, the exposed user u updated its secrets and, second, user $\overline{u}$ updates its secrets accordingly (which can happen after it receives the next message from u). But if the randomness was exposed, then secrets would need to be updated at least twice until the security is restored.

Exposing user u with $\mathsf{rand} =$ "send" additionally allows the attack shown in $\mathcal{D}_6$. The adversary exposes the state and the sending randomness of $\mathcal{I}$, encrypts 1 locally using these exposed values of $\mathcal{I}$, and then calls SEND to get a challenge ciphertext sent by $\mathcal{I}$. The adversary compares whether the two ciphertexts are the same to determine the secret bit. We have $\mathsf{Adv}^{\mathsf{inter}}_{\mathsf{Ch}}(\mathcal{D}_6) = 1$. More broadly, if the adversary exposes the secrets of u with $\mathsf{rand} =$ "send" it will always be able to tell what is the next message encrypted by u.

Exposing with $\mathsf{rand} =$ "recv" does not generically endow the adversary with the ability to do any additional attacks.

2.4.2 Optimal security of a channel

Our full security game is obtained by adding a minimal amount of code to INTER to disallow the generic attacks just discussed. Consider the game AEAC (authenticated encryption against compromise) shown in Fig. 2.9. We define the advantage of an adversary $\mathcal{D}$ against channel Ch by $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{Ch}}(\mathcal{D}) = 2\Pr[\mathrm{AEAC}^{\mathcal{D}}_{\mathsf{Ch}}] - 1$.

We now have a total of eight variables to control the behavior of the adversary and prevent it from abusing trivial attacks. Some of the variables are summarized in Table 2.1. We have already seen s_u, r_u, nextop, and $\mathbf{ctable}_u$ in INTER. The new variables we have added in AEAC are tables $\mathbf{forge}_u$ and $\mathbf{ch}_u$, number $\mathcal{X}_u \in \mathbb{N}$, and flag $\mathsf{restricted}_u \in \{\mathsf{true}, \mathsf{false}\}$. We now discuss the new variables.

$\underline{\text{Game AEAC}_{\mathsf{Ch}}^{\mathcal{D}}}$

$b \leftarrow_{\$} \{0,1\}$; $s_{\mathcal{I}} \leftarrow 0$; $r_{\mathcal{I}} \leftarrow 0$; $s_{\mathcal{R}} \leftarrow 0$; $r_{\mathcal{R}} \leftarrow 0$; $\mathsf{restricted}_{\mathcal{I}} \leftarrow \mathsf{false}$; $\mathsf{restricted}_{\mathcal{R}} \leftarrow \mathsf{false}$

$\mathbf{forge}_{\mathcal{I}}[0\ldots\infty] \leftarrow$ "nontrivial" ; $\mathbf{forge}_{\mathcal{R}}[0\ldots\infty] \leftarrow$ "nontrivial" ; $\mathcal{X}_{\mathcal{I}} \leftarrow 0$; $\mathcal{X}_{\mathcal{R}} \leftarrow 0$

$(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow_{\$} \mathsf{Ch.Init}$; $(z_{\mathcal{I}}, z_{\mathcal{R}}) \leftarrow_{\$} (\mathsf{Ch.SendRS})^2$; $(\eta_{\mathcal{I}}, \eta_{\mathcal{R}}) \leftarrow_{\$} (\mathsf{Ch.RecvRS})^2$

$b' \leftarrow_{\$} \mathcal{D}^{\textsc{Send},\textsc{Recv},\textsc{Exp}}$

Return $(b' = b)$

$\underline{\textsc{Send}(u, m_0, m_1, ad)}$ **/** $u \in \{\mathcal{I}, \mathcal{R}\}, (m_0, m_1, ad) \in (\{0,1\}^*)^3$

If $\mathsf{nextop} \neq (u, \text{"send"})$ and $\mathsf{nextop} \neq \bot$ then return $\bot$

If $|m_0| \neq |m_1|$ then return $\bot$

If ($r_u < \mathcal{X}_u$ or $\mathsf{restricted}_u$ or $\mathbf{ch}_u[s_u + 1] =$ "forbidden") and $m_0 \neq m_1$ then return $\bot$

$(st_u, c) \leftarrow \mathsf{Ch.Send}(st_u, ad, m_b; z_u)$

$\mathsf{nextop} \leftarrow \bot$; $s_u \leftarrow s_u + 1$; $z_u \leftarrow_{\$} \mathsf{Ch.SendRS}$

If not $\mathsf{restricted}_u$ then $\mathbf{ctable}_{\overline{u}}[s_u] \leftarrow (c, ad)$

If $m_0 \neq m_1$ then $\mathbf{ch}_u[s_u] \leftarrow$ "done"

Return c

$\underline{\textsc{Recv}(u, c, ad)}$ **/** $u \in \{\mathcal{I}, \mathcal{R}\}, (c, ad) \in (\{0,1\}^*)^2$

If $\mathsf{nextop} \neq (u, \text{"recv"})$ and $\mathsf{nextop} \neq \bot$ then return $\bot$

$(st_u, m) \leftarrow \mathsf{Ch.Recv}(st_u, ad, c; \eta_u)$

$\mathsf{nextop} \leftarrow \bot$; $\eta_u \leftarrow_{\$} \mathsf{Ch.RecvRS}$

If $m = \bot$ then return $\bot$

$r_u \leftarrow r_u + 1$

If $\mathbf{forge}_u[r_u] =$ "trivial" and $(c, ad) \neq \mathbf{ctable}_u[r_u]$ then $\mathsf{restricted}_u \leftarrow \mathsf{true}$

If $\mathsf{restricted}_u$ or ($b = 0$ and $(c, ad) \neq \mathbf{ctable}_u[r_u]$) then return m

Return $\bot$

$\underline{\textsc{Exp}(u, \mathsf{rand})}$ **/** $u \in \{\mathcal{I}, \mathcal{R}\}, \mathsf{rand} \in \{\varepsilon, \text{"send"}, \text{"recv"}\}$

If $\mathsf{nextop} \neq \bot$ then return $\bot$

If $\mathsf{restricted}_u$ then return (st_u, z_u, η_u)

If $\exists i \in (r_u, s_{\overline{u}}]$ s.t. $\mathbf{ch}_{\overline{u}}[i] =$ "done" then return $\bot$

$\mathbf{forge}_{\overline{u}}[s_u + 1] \leftarrow$ "trivial" ; $(z, \eta) \leftarrow (\varepsilon, \varepsilon)$; $\mathcal{X}_{\overline{u}} \leftarrow s_u + 1$

If $\mathsf{rand} =$ "send" then

  $\mathsf{nextop} \leftarrow (u, \text{"send"})$; $z \leftarrow z_u$; $\mathcal{X}_{\overline{u}} \leftarrow s_u + 2$

  $\mathbf{forge}_{\overline{u}}[s_u + 2] \leftarrow$ "trivial" ; $\mathbf{ch}_u[s_u + 1] \leftarrow$ "forbidden"

Else if $\mathsf{rand} =$ "recv" then

  $\mathsf{nextop} \leftarrow (u, \text{"recv"})$; $\eta \leftarrow \eta_u$

Return (st_u, z, η)

Figure 2.9. Game defining AEAC security of channel Ch.

The table $\mathbf{forge}_u$ was added to prevent the type of attack shown in $\mathcal{D}_3$. When the adversary calls Exp on user u we set $\mathbf{forge}_{\overline{u}}$ to "trivial" for the indices of ciphertexts for which this adversary is now necessarily able to create forgeries. If the adversary takes advantage of this

Table 2.1. Table summarizing some important variables in game AEAC. A "$-$" indicates a way in which the behavior of the adversary is being restricted. A "$+$" indicates a way in which the behavior of the adversary is being enabled.

Variable	Set to x when y occurs	Effect
nextop_u	$(u,$"send"$)$ when z_u is exposed	$-$ u must send next
	$(u,$"recv"$)$ when η_u is exposed	$-$ u must receive next
$\mathbf{forge}_u$	"trivial" when $\overline{u}$ is exposed	$-$ forgeries to u set $\mathsf{restricted}_u$
$\mathbf{ch}_u$	"done" when challenge from u	$-$ prevents an exposure of $\overline{u}$
	"forbidden" when z_u is exposed	$-$ prevents a challenge from u
$\mathcal{X}_u$	when $\overline{u}$ is exposed	$-$ prevents challenges until $r_u = \mathcal{X}_u$
$\mathsf{restricted}_u$	true when trivial forger y to u	$-$ prevents challenges from u $+$ (c, ad) from u not added to $\mathbf{ctable}_{\overline{u}}$ $\pm$ show decryption of (c, ad) sent to u $+$ EXP calls to u always allowed and will not change other variables

to send a ciphertext of its own creation to $\overline{u}$ then the flag $\mathsf{restricted}_{\overline{u}}$ will be set, whose effect we will describe momentarily.

The table $\mathbf{ch}_u$ is used to prevent the types of attacks shown by $\mathcal{D}_4$ and $\mathcal{D}_6$. Whenever the adversary makes a valid challenge query[2] to user u we set $\mathbf{ch}_u[s_u]$ to "done". The game will not allow the adversary to expose $\overline{u}$'s secrets if there are any challenge queries for which u sent a ciphertext that $\overline{u}$ has not received yet. This use of $\mathbf{ch}_u$ prevents an attack like $\mathcal{D}_4$. To prevent an attack like $\mathcal{D}_6$, we set $\mathbf{ch}_u[s_u + 1]$ to "forbidden" whenever the adversary exposes the state and sending randomness of u. This disallows the adversary from doing a challenge query during its next SEND call to u (the call for which the adversary knows the corresponding randomness).

The number $\mathcal{X}_u$ prevents attacks like $\mathcal{D}_5$. When u is exposed $\mathcal{X}_{\overline{u}}$ will be set to a number that is 1 or 2 greater than the current number of ciphertexts u has sent (depending on the value of rand) and challenge queries from $\overline{u}$ will not be allowed until it has received that many ciphertexts. This ensures that the challenge queries from $\overline{u}$ are not issued with respect to exposed keys of u.[3]

Finally the flag $\mathsf{restricted}_u$ serves to both allow and disallow some attacks. The flag is

[2] We use the term challenge query to refer to a SEND query for which $m_0 \neq m_1$.

[3] The symbol $\mathcal{X}$ (chi) is meant to evoke the word "challenge" because it stores the next time the adversary may make a challenge query.

initialized to false. It is set to true when the adversary forges a ciphertext to u after exposing $\overline{u}$. Once u has received a different ciphertext than was sent by $\overline{u}$ there is no reason to think that u should be able to decrypt ciphertexts sent by $\overline{u}$ or send its own ciphertexts to $\overline{u}$. As such, if u is restricted (i.e. $\mathsf{restricted}_u = \mathsf{true}$) we will not add its ciphertexts to $\mathbf{ctable}_{\overline{u}}$, we will always show the true output when u attempts to decrypt ciphertexts given to it by the adversary (even if they were sent by $\overline{u}$), and if the adversary asks to expose u we will return all of its secret state without setting any of the other variables that would restrict the actions the adversary is allowed to take.

The above describes how $\mathsf{restricted}_u$ allows some attacks. Now we discuss how it prevents attacks like $\mathcal{D}_{3.1}$ and $\mathcal{D}_{3.2}$. Once the adversary has sent its own ciphertext to u we must assume that the adversary will be able to decrypt ciphertexts sent by u and able to send its own ciphertexts to u that will decrypt to non-$\perp$ values. The adversary could simply have "replaced" $\overline{u}$ with itself. To address this we prevent all challenge queries from u, and decryptions performed by u are always given back to the adversary regardless of the secret bit.

Informal description of the security game.

In Section 2.4.3 we provide a thorough written description of our security model to facilitate high-level understanding of it. For intricate security definitions like ours there is often ambiguity or inconsistency in subtle corner cases of the definition when written out fully in text. As such this description should merely be considered an informal aid while the pseudocode of Fig. 2.9 is the actual definition.

Comparison to recent definitions.

The three recents works we studied while deciding how to write our security definition were [29], [17], and [43]. Their settings were all distinct, but each presented security models that involve different "stages" of keys. All three works made distinct decisions in how to address challenges in different stages. In Section 2.6 we discuss these decisions, noting that they result in qualitatively identical but quantitatively distinct definitions.

2.4.3 Informal description of our security definition

We now attempt to provide an informal description of our security definition to facilitate high-level understanding of it. The security experiment starts by choosing a challenge bit b. The channel's initialization algorithm $\mathsf{Ch.Init}$ is run to produce the initial state of users $\mathcal{I}$ and $\mathcal{R}$. Then the adversary is run and given the ability to ask either user to send a message or receive a ciphertext, or to expose their state. The adversary is allowed to perform these actions in essentially any order with a small restriction that we describe momentarily. Eventually the adversary must halt and output a bit b'. If $b' = b$, then the adversary is considered to have won, otherwise it has lost.

We first describe sending of messages and receiving of ciphertexts, without reference to the restrictions that will be placed on these operations after the exposure of a user's secrets. When the adversary asks user u to send a message, it will provide two messages m_0 and m_1 of equal length, together with associated data ad. Then algorithm $\mathsf{Ch.Send}$ will be run on input the state of u, m_b, and ad. The resulting ciphertext will be returned to the adversary. When the adversary asks user u to receive a ciphertext, it will provide the ciphertext c and associated data ad. Then algorithm $\mathsf{Ch.Recv}$ will be run on input the state of u, c, and ad. The adversary gets back $\perp$ if the decryption of c with ad helps adversary to trivially win the game (i.e. it is identical to the ciphertext sent by $\overline{u}$, and only if the last ciphertext sent by $\overline{u}$ was already accepted by u). If the decryption of the provided ciphertext failed, the adversary gets $\perp$ as the result. Otherwise the actual decryption will be returned if $b = 0$, and $\perp$ will be returned if $b = 1$.

We proceed to describe the effect of exposing a user, which introduces the majority of the complexity of our model. When exposing the current state of user u the adversary will provide an additional string rand that can equal ε, "send", or "recv". The flag rand indicates whether the adversary is exposing the state of the user while they are "at rest", while they are sending a message, or while they are receiving a ciphertext. In the latter two cases the adversary is required to follow up with a valid request that the specified user performs the specified action,

and the adversary will be proactively given back the randomness that will be used when this action is performed. Additionally, the expose query to u is not allowed if it allows the adversary to trivially win the game, which happens if $\overline{u}$ was asked to send a message to u (encrypting one of two challenge plaintexts) that was not yet received by u.

After u's state is exposed, the adversary cannot ask $\overline{u}$ to send one of two different messages until $\overline{u}$ has received the *next* ciphertext that u sends. If u was exposed while they were sending a message then this refers to the next ciphertext after the one that is about to be sent, and the adversary is further not allowed to ask u to send one of two different messages for the current sending operation. These are minimal necessary requirements that give user u a chance to recover from exposure.

If the adversary exposes u and uses the exposed secrets to forge a ciphertext to $\overline{u}$, then $\overline{u}$ becomes "restricted," which we will describe in the next paragraph. More precisely, the condition is that the ciphertext that adversary sends to $\overline{u}$ is accepted after $\overline{u}$ has received the last ciphertext that was sent by u (prior to u's state getting exposed) but before $\overline{u}$ has received the next ciphertext that u sends. If u is exposed while sending a message the next ciphertext just mentioned refers to the next ciphertext *after* the one that is about to be sent.

When a user u is restricted a number of the statements we made above no longer hold. When a restricted user is exposed, their current state, next randomness for sending, and next randomness for receiving will be freely given to the adversary without placing any restrictions on its future actions. Similarly, there is no longer any restriction on when u can be exposed (except if $\overline{u}$ is not restricted then u cannot be exposed after $\overline{u}$ was exposed while performing an operation, until $\overline{u}$ has finished performing the corresponding operation). The adversary is no longer allowed to ask u to send one of two different messages. When u is asked to receive a ciphertext, the adversary will only be given back $\perp$ if the decryption failed, meaning the adversary always gets back the actual output of $\mathsf{Ch.Recv}$ (no other conditions will trigger $\perp$ to be returned). Finally, a query asking $\overline{u}$ to accept a ciphertext no longer always returns $\perp$ if this ciphertext was produced by querying a restricted user u to send a message. Instead, if the previous message sent from u

was accepted by $\overline{u}$, then the above query returns the actual output of $\mathsf{Ch.Recv}$ if $b = 0$, and $\perp$ otherwise.

This concludes the description of our security model. The above description already contains some ambiguity we are aware of in situations where a user sends a ciphertext that happens to be identical to a ciphertext it has already sent. We do not attempt to resolve this ambiguity in text, but again refer the reader to our pseudocode for the precise definition.

2.5 Construction of a secure channel

2.5.1 Our construction

We are not aware of any secure channels that would meet (or could easily be modified to meet) our security notion. The "closest" (for some unspecified, informal notion of distance) is probably the Signal Double Ratchet Algorithm. However, it relies on symmetric authenticated encryption for both privacy and integrity so it is inherently incapable of achieving our strong notion of security. Later, we describe an attack against a variant of our proposed construction that uses symmetric primitives to exhibit the sorts of attacks that are unavoidable when using them. A straightforward variant of this attack would also apply against the Double Ratchet Algorithm.

In this section we construct our cryptographic channel and motivate our design decisions by giving attacks against variants of the channel. In Section 2.5.2 we will prove its security by reducing it to that of its underlying components.

The idea of our scheme is as follows. Both parties will keep track of a transcript of the messages they have sent and received, τ_s and τ_r. These will be included as a part of every ciphertext and verified before a ciphertext is accepted. On seeing a new ciphertext the appropriate transcript is updated to be the hash of the ciphertext (note that the old transcript is part of this ciphertext, so the transcript serves as a record of the entire conversation). Sending transcripts (vector of τ_s) are stored until the other party has acknowledged receiving a more recent transcript.

For authenticity, every time a user sends a ciphertext they authenticate it with a digital

signature and include in it the verification key for the signing key that they will use to sign the next ciphertext they send. Any time a user receives a ciphertext they will use the new receiving transcript produced to update their current signing key.

For privacy, messages will be encrypted using public-key encryption. With every ciphertext the sender will include the encryption key for a new decryption key they have generated. Decryption keys are stored until the other party has acknowledged receiving a more recent encryption key. Any time a user receives a ciphertext they will use the new receiving transcript to produced to update each of these keys. The encryption will use as a label all of the extra data that will be included with the ciphertext (i.e. a sending counter, a receiving counter, an associated data string, a new verification key, a new encryption key, a receiving transcript, and a sending transcript). The formal definition of our channel is as follows.

Cryptographic channel $\mathsf{SCH}[\mathsf{DS},\mathsf{PKE},\mathsf{H}]$.

Let DS be a key-updatable digital signature scheme, PKE be a key-updatable public-key encryption scheme, and H be a family of functions. We build a cryptographic channel $\mathsf{SCh} = \mathsf{SCH}[\mathsf{DS},\mathsf{PKE},\mathsf{H}]$ as defined in Fig. 2.10.

A user's state st_u, among other values, contains counters s_u, r_u, r_u^{ack}. Here, s_u is the number of messages that u sent to $\overline{u}$, and r_u is the number of messages they received back from $\overline{u}$. The counter r_u^{ack} stores the last value of $r_{\overline{u}}$ in a ciphertext received by u (i.e. the index of the last ciphertext that u believes $\overline{u}$ has received and acknowledged). This counter is used to ensure that prior to running a signature verification algorithm, the vertification key vk is updated with respect to the same transcripts as the signing key sk (at the time it was used to produce the signature). Note that algorithm $\mathsf{DS.Vrfy}$ returns (vk'', t) where t is the result of verifying that σ is a valid signature for v with respect to verification key vk'' (using the notation convention from Section 2.2).

Algorithm $\mathsf{SCh.Init}$

$(sk_{\mathcal{I}}, vk_{\mathcal{R}}) \leftarrow\$ \mathsf{DS.Kg}\,;\ (ek_{\mathcal{I}}, \mathbf{dk}_{\mathcal{R}}[0]) \leftarrow\$ \mathsf{PKE.Kg}$
$(sk_{\mathcal{R}}, vk_{\mathcal{I}}) \leftarrow\$ \mathsf{DS.Kg}\,;\ (ek_{\mathcal{R}}, \mathbf{dk}_{\mathcal{I}}[0]) \leftarrow\$ \mathsf{PKE.Kg}$
$hk \leftarrow\$ \mathsf{H.Kg}\,;\ \tau_r \leftarrow \varepsilon\,;\ \tau_s[0] \leftarrow \varepsilon\,;\ s \leftarrow r \leftarrow r^{ack} \leftarrow 0$
$st_{\mathcal{I}} \leftarrow (s, r, r^{ack}, sk_{\mathcal{I}}, vk_{\mathcal{I}}, ek_{\mathcal{I}}, \mathbf{dk}_{\mathcal{I}}, hk, \tau_r, \tau_s)$
$st_{\mathcal{R}} \leftarrow (s, r, r^{ack}, sk_{\mathcal{R}}, vk_{\mathcal{R}}, ek_{\mathcal{R}}, \mathbf{dk}_{\mathcal{R}}, hk, \tau_r, \tau_s)$
Return $(st_{\mathcal{I}}, st_{\mathcal{R}})$

Algorithm $\mathsf{SCh.Send}(\vec{k}, ad, m)$

$(s, r, r^{ack}, sk, vk, ek, \mathbf{dk}, hk, \tau_r, \tau_s) \leftarrow \vec{k}\,;\ s \leftarrow s+1$
$(sk', vk') \leftarrow\$ \mathsf{DS.Kg}\,;\ (ek', \mathbf{dk}[s]) \leftarrow\$ \mathsf{PKE.Kg}$
$\ell \leftarrow (s, r, ad, vk', ek', \tau_r, \tau_s[s-1])$
$(ek', c') \leftarrow\$ \mathsf{PKE.Enc}(ek, \ell, m, \tau_s[r^{ack}+1, \ldots, s-1])$
$v \leftarrow (c', \ell)\,;\ \sigma \leftarrow\$ \mathsf{DS.Sign}(sk, v)$
$c \leftarrow (\sigma, v)\,;\ \tau_s[s] \leftarrow \mathsf{H.Ev}(hk, c)$
$\vec{k} \leftarrow (s, r, r^{ack}, sk', vk, ek, \mathbf{dk}, hk, \tau_r, \tau_s)$
Return $(\vec{k}, c)$

Algorithm $\mathsf{SCh.Recv}(\vec{k}, ad, c)$

$(s, r, r^{ack}, sk, vk, ek, \mathbf{dk}, hk, \tau_r, \tau_s) \leftarrow \vec{k}$
$(\sigma, v) \leftarrow c\,;\ (c', \ell) \leftarrow v$
$(s', r', ad', vk', ek', \tau_r', \tau_s') \leftarrow \ell$
If $s' \neq r+1$ or $\tau_r' \neq \tau_s[r']$ or $\tau_s' \neq \tau_r$ or $ad \neq ad'$ then return $(\vec{k}, \bot)$
$(vk'', t) \leftarrow \mathsf{DS.Vrfy}(vk, \sigma, v, \tau_s[r^{ack}+1, \ldots, r'])$
If not t then return $(\vec{k}, \bot)$
$r \leftarrow r+1\,;\ r^{ack} \leftarrow r'\,;\ m \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}[r^{ack}], \ell, c')$
$\tau_s[0, \ldots, r^{ack}-1] \leftarrow \bot\,;\ \mathbf{dk}[0, \ldots, r^{ack}-1] \leftarrow \bot$
$\tau_r \leftarrow \mathsf{H.Ev}(hk, c)\,;\ sk \leftarrow\$ \mathsf{DS.UpdSk}(sk, \tau_r)$
For $i \in [r^{ack}, s_u]$ do $\mathbf{dk}[i] \leftarrow\$ \mathsf{PKE.UpdDk}(\mathbf{dk}[i], \tau_r)$
$\vec{k} \leftarrow (s, r, r^{ack}, sk, vk', ek', \mathbf{dk}, hk, \tau_r, \tau_s)$
Return $(\vec{k}, m)$

Figure 2.10. Construction of channel $\mathsf{SCh} = \mathsf{SCH}[\mathsf{DS}, \mathsf{PKE}, \mathsf{H}]$ from function family H, key-updatable digital signature scheme DS, and key-updatable public-key encryption scheme PKE.

Inefficiencies of SCh**.**

A few aspects of SCh are less efficient than one would a priori hope. The state maintained by a user u (specifically the tables $\mathbf{dk}_u$ and $\tau_{s,u}$) is not constant in size, but instead grows linearly with the number of ciphertexts that u sent to $\overline{u}$ without receiving a reply back. Additionally, when DS is instantiated with the particular choice of DS that we define in Section 2.8 the length of the ciphertext sent by a user u and the amount of state it stores will grow linearly in the number of

ciphertexts that u has received since the last time they sent a ciphertext. Such inefficiencies would be unacceptable for a protocol like TLS or SSH, but in our motivating context of messaging is it plausible that they are acceptable. Each message is human generated and the state gets "refreshed" regularly if the two users regularly reply to one another. One could additionally consider designing an app to regularly send an empty message whose sole purpose is state refreshing. We leave as interesting future work improving on the efficiency of our construction.

Design decisions.

We will now discuss attacks against different variants of SCh. This serves to motivate the decisions made in its design and give intuition for why it achieves the desired security. Several steps in the security proof of this construction can be understood by noting which of these attacks are ruled out in the process.

The attacks are shown in Fig. 2.11 and Fig. 2.12. The first several attacks serve to demonstrate that $\mathsf{Ch.Send}$ must use a sufficient amount of randomness (shown in $\mathcal{D}_a$, $\mathcal{D}_b$, $\mathcal{D}_c$) and that H needs to be collision resistant (shown in $\mathcal{D}_b$, $\mathcal{D}_c$). The next attack shows why our construction would be insecure if we did not use labels with PKE (shown in $\mathcal{D}_d$). Then we provide two attacks showing why the keys of DS and PKE need to be updated (shown in $\mathcal{D}_e$, $\mathcal{D}_f$). Then we show an attack that arises if multiple valid signatures can be found for the same string (shown in $\mathcal{D}_g$). Finally, we conclude with attacks that would apply if we used symmetric instead of asymmetric primitives to build SCh (shown in $\mathcal{D}_h$, $\mathcal{D}_i$).

Scheme with insufficient sending entropy.

Any scheme whose sending algorithm has insufficient entropy will necessarily be insecure. For simplicity let SCh_1 be a variant of SCh such that $\mathsf{SCh}_1.\mathsf{Send}$ is deterministic (the details of how we are making it deterministic do not matter). We can attack both the message privacy and the integrity of such a scheme.

Consider the adversary $\mathcal{D}_a$. It exposes $\mathcal{I}$, encrypts the message 1 locally, and then sends a challenge query to $\mathcal{I}$ asking for the encryption of either 1 or 0. By comparing the ciphertext it

Adversary $\mathcal{D}_a^{\text{SEND,RECV,EXP}}$	Adversary $\mathcal{D}_c^{\text{SEND,RECV,EXP}}$
$(\vec{k}, z, \eta) \leftarrow \text{EXP}(\mathcal{I}, \varepsilon)$ $(\vec{k}, c) \leftarrow \mathsf{SCh}_1.\mathsf{Send}(\vec{k}, \varepsilon, 1)$ $c' \leftarrow \text{SEND}(\mathcal{I}, 0, 1, \varepsilon)$ If $c = c'$ then return 1 Return 0 **Adversary $\mathcal{D}_b^{\text{SEND,RECV,EXP}}$** $(\vec{k}, z, \eta) \leftarrow \text{EXP}(\mathcal{I}, \varepsilon)$ $(\vec{k}, c) \leftarrow\$ \mathsf{Ch}.\mathsf{Send}(\vec{k}, \varepsilon, 1)$ $m \leftarrow \text{RECV}(\mathcal{R}, c, \varepsilon)$ $c \leftarrow \text{SEND}(\mathcal{I}, 1, 1, \varepsilon)$ $c \leftarrow \text{SEND}(\mathcal{R}, 1, 1, \varepsilon)$ $m \leftarrow \text{RECV}(\mathcal{I}, c, \varepsilon)$ If $m = \bot$ then return 1 Return 0	$(\vec{k}, z, \eta) \leftarrow \text{EXP}(\mathcal{I}, \varepsilon)$ $(\vec{k}, c) \leftarrow\$ \mathsf{Ch}.\mathsf{Send}(\vec{k}, \varepsilon, 1)$ $m \leftarrow \text{RECV}(\mathcal{R}, c, \varepsilon)$ $c \leftarrow \text{SEND}(\mathcal{I}, 1, 1, \varepsilon)$ $(\vec{k}, z, \eta) \leftarrow \text{EXP}(\mathcal{R}, \varepsilon)$ $(\vec{k}, c) \leftarrow\$ \mathsf{Ch}.\mathsf{Send}(\vec{k}, \varepsilon, 1)$ $m \leftarrow \text{RECV}(\mathcal{I}, c, \varepsilon)$ If $m = \bot$ then return 1 else return 0 **Adversary $\mathcal{D}_d^{\text{SEND,RECV,EXP}}$** $(\vec{k}, z, \eta) \leftarrow \text{EXP}(\mathcal{I}, \varepsilon)$ $c \leftarrow \text{SEND}(\mathcal{I}, 0, 1, \varepsilon)$ $(s, r, r^{ack}, sk, vk, ek, \mathbf{dk}, hk, \tau_r, \tau_s) \leftarrow \vec{k}$ $(\sigma, (c', (s, r, ad, vk', ek', \tau_r, \tau_s))) \leftarrow c$ $v \leftarrow (c', (s, r, 1^{128}, vk', ek', \tau_r, \tau_s))$ $\sigma \leftarrow\$ \mathsf{DS}.\mathsf{Sign}(sk, v)$ $m \leftarrow \text{RECV}(\mathcal{R}, (\sigma, v), 1^{128})$ If $m = 1$ then return 1 else return 0

Figure 2.11. Attacks against variants of SCh.

produced to the one returned by SEND it can determine which message was encrypted, learning the secret bit. We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_1}(\mathcal{D}_a) = 1$. This attack is fairly straightforward and will be ruled out by the security of PKE in our proof without having to be addressed directly.

The attacks against integrity are more subtle. They are explicitly addressed in the first game transition of our proof. Let $\mathsf{Ch} = \mathsf{SCh}_1$ and consider adversaries $\mathcal{D}_b$ and $\mathcal{D}_c$. They both start by doing the same sequence of operations: expose $\mathcal{I}$, use its secret state to encrypt and send message 1 to $\mathcal{R}$, then ask $\mathcal{I}$ to produce an encryption of 1 for $\mathcal{R}$ (which will be the same ciphertext as above, because SCh_1.Send is deterministic). Now $\mathsf{restricted}_{\mathcal{R}} = \mathsf{true}$ because oracle RECV was called on a trivially fogeable ciphertext that was not produced by oralce SEND. But $\mathcal{R}$ has received the exact same ciphertext that $\mathcal{I}$ sent. Different attacks are possible from this point.

Adversary $\mathcal{D}_b$ just asks $\mathcal{R}$ to send a message and forwards it along to $\mathcal{I}$. Since $\mathcal{R}$ was

restricted the ciphertext does not get added to $\mathbf{ctable}_{\mathcal{I}}$ so it can be used to discover the secret bit. We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_1}(\mathcal{D}_b) = 1$. Adversary $\mathcal{D}_c$ exposes $\mathcal{R}$ and uses the state it obtains to create its own forgery to $\mathcal{I}$. It then returns 1 or 0 depending on whether RECV returns the correct decryption or $\perp$. This attack succeeds because exposing $\mathcal{R}$ when it is restricted will not set any of the variables that would typically prevent the adversary from winning by creating a forgery. We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_1}(\mathcal{D}_c) = 1$. We have not shown it, but another message privacy attack at this point (instead of proceeding as $\mathcal{D}_b$ or $\mathcal{D}_c$) could have asked for another challenge query from $\mathcal{I}$, exposed $\mathcal{R}$, and used the exposed state to trivially determine which message was encrypted.

Scheme without collision-resistant hashing.

If it is easy to find collisions in H then we can attack the channel by causing both parties to have matching transcripts despite having seen different sequences of ciphertexts. For concreteness let SCh_2 be a variant of our scheme using a hash function that outputs 0^{128} on all inputs. Let $\mathsf{Ch} = \mathsf{SCh}_2$ and again consider adversaries $\mathcal{D}_b$ and $\mathcal{D}_c$. We no longer expect the ciphertexts that they produce locally to match the ciphertexts returned by $\mathcal{I}$. However, they will have the same hash value and thus produce the same transcript $\tau_{r,\mathcal{R}} = 0^{128} = \tau_{s,\mathcal{I}}$. Consequently, $\mathcal{R}$ still updates its signing key in the same way regardless of whether it receives the ciphertext produced by $\mathcal{I}$ or the ciphertext locally generated by adversary. So the messages subsequently sent by $\mathcal{R}$ will still be accepted by $\mathcal{I}$. We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_2}(\mathcal{D}_b) = 1$ and $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_2}(\mathcal{D}_c) = 1$.

Scheme without PKE labels.

Let SCh_3 be a variant of SCh that uses a public-key encryption scheme that does not accept labels and consider adversary $\mathcal{D}_d$. It exposes $\mathcal{I}$ and asks $\mathcal{I}$ for a challenge query. It then uses the state it exposed to trivially modify the ciphertext sent from $\mathcal{I}$ (we chose to have it change ad from ε to 1^{128}) and sends it to $\mathcal{R}$. Since the ciphertext sent to $\mathcal{R}$ has different associated data than the one sent by $\mathcal{I}$ the adversary will be given the decryption of this ciphertext. But without the use of labels this decryption by PKE is independent of the associated data and will thus reveal the true decryption of the challenge ciphertext to $\mathcal{I}$. We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_3}(\mathcal{D}_d) = 1$.

Adversary $\mathcal{D}_e^{\text{SEND},\text{RECV},\text{EXP}}$

$(\vec{k}, z, \eta) \leftarrow \text{EXP}(\mathcal{I}, \varepsilon)$
$c_{\mathcal{I}} \leftarrow \text{SEND}(\mathcal{I}, 1, 1, \varepsilon)$
$(\sigma, (c', (s, r, ad, vk_{\mathcal{I}}, ek_{\mathcal{I}}, \tau_r, \tau_s))) \leftarrow c_{\mathcal{I}}$
$(\vec{k}, c) \leftarrow\$ \mathsf{SCh}_4.\mathsf{Send}(\vec{k}, \varepsilon, 0)$
$m \leftarrow \text{RECV}(\mathcal{R}, c, \varepsilon)$
$(\vec{k}, z, \eta) \leftarrow \text{EXP}(\mathcal{R}, \varepsilon)$
$(s, r, r^{ack}, sk, vk, ek, \mathbf{dk}, hk, \tau_r, \tau_s) \leftarrow \vec{k}$
$\tau_r \leftarrow \mathsf{H.Ev}(hk, c_{\mathcal{I}})$
$\vec{k} \leftarrow (s, r, r^{ack}, sk, vk_{\mathcal{I}}, ek_{\mathcal{I}}, \mathbf{dk}, hk, \tau_r, \tau_s)$
$(\vec{k}, c) \leftarrow\$ \mathsf{SCh}_4.\mathsf{Send}(\vec{k}, \varepsilon, 1)$
$m \leftarrow \text{RECV}(\mathcal{I}, c, \varepsilon)$
If $m = \perp$ then return 1 else return 0

Adversary $\mathcal{D}_f^{\text{SEND},\text{RECV},\text{EXP}}$

$(\vec{k}, z, \eta) \leftarrow \text{EXP}(\mathcal{I}, \varepsilon)$
$(\sigma, (c', \ell)) \leftarrow \text{SEND}(\mathcal{I}, 0, 1, \varepsilon)$
$(\vec{k}, c) \leftarrow\$ \mathsf{SCh}_5.\mathsf{Send}(\vec{k}, \varepsilon, 0)$
$m \leftarrow \text{RECV}(\mathcal{R}, c, \varepsilon)$
$(\vec{k}, z, \eta) \leftarrow \text{EXP}(\mathcal{R}, \varepsilon)$
$(s, r, r^{ack}, sk, vk, ek, \mathbf{dk}, hk, \tau_r, \tau_s) \leftarrow \vec{k}$
$m \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}[0], \ell, c')$
If $m = 1$ then return 1 else return 0

Adversary $\mathcal{D}_g^{\text{SEND},\text{RECV},\text{EXP}}$

$(\vec{k}, z, \eta) \leftarrow \text{EXP}(\mathcal{I}, \varepsilon)$
$(\sigma, v) \leftarrow \text{SEND}(\mathcal{I}, 0, 1, \varepsilon)$
$(s, r, r^{ack}, sk, vk, ek, \mathbf{dk}, hk, \tau_r, \tau_s) \leftarrow \vec{k}$
$m \leftarrow \text{RECV}(\mathcal{R}, (\sigma \,\|\, sk, v), \varepsilon)$
If $m = 1$ then return 1
Return 0

Adversary $\mathcal{D}_h^{\text{SEND},\text{RECV},\text{EXP}}$

$(\vec{k}, z, \eta) \leftarrow \text{EXP}(\mathcal{I}, \varepsilon)$
$(s, r, r^{ack}, sk, vk, ek, \mathbf{dk}, hk, \tau_r, \tau_s) \leftarrow \vec{k}$
$\vec{k} \leftarrow (s, r, r^{ack}, vk, sk, ek, \mathbf{dk}, hk, \tau_r, \tau_s)$
$(\vec{k}, c) \leftarrow\$ \mathsf{SCh}_7.\mathsf{Send}(\vec{k}, \varepsilon, 0)$
$m \leftarrow \text{RECV}(\mathcal{I}, c, \varepsilon)$
If $m = \perp$ then return 1
Return 0

Adversary $\mathcal{D}_i^{\text{SEND},\text{RECV},\text{EXP}}$

$(\vec{k}, z, \eta) \leftarrow \text{EXP}(\mathcal{I}, \varepsilon)$
$(\sigma, (c', \ell)) \leftarrow \text{SEND}(\mathcal{I}, 0, 1, \varepsilon)$
$(s, r, r^{ack}, sk, vk, ek, \mathbf{dk}, hk, \tau_r, \tau_s) \leftarrow \vec{k}$
$m \leftarrow \mathsf{PKE.Dec}(ek, \ell, c')$
If $m = 1$ then return 1
Return 0

Figure 2.12. Attacks against variants of SCh.

Schemes without updatable keys.

We will now show why it is necessary to define new forms of PKE and DS for our construction.

Let SCh_4 be a variant of SCh that uses a digital signature scheme that does not update its keys. Consider adversary $\mathcal{D}_e$. It exposes $\mathcal{I}$, then queries SEND for $\mathcal{I}$ to send a message to $\mathcal{R}$, but uses the exposed secrets to replace it with a locally produced ciphertext c. It calls RECV for $\mathcal{R}$ with c, which sets $\mathsf{restricted}_{\mathcal{R}} = \mathsf{true}$. Since the signing key is not updated in SCh_4, the adversary now exposes $\mathcal{R}$ to obtain a signing key whose signatures will be accepted by $\mathcal{I}$. It uses this to forge a ciphertext to $\mathcal{I}$ to learn the secret bit. We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_4}(\mathcal{D}_e) = 1$.

Let SCh_5 be a variant of SCh that uses a public-key encryption scheme that does not

update its keys. Consider adversary $\mathcal{D}_f$. It exposes $\mathcal{I}$ and uses this to send $\mathcal{R}$ a different ciphertext than is sent by $\mathcal{I}$ (setting $\mathsf{restricted}_{\mathcal{R}} = \mathsf{true}$). Since the decryption key is not updated, the adversary now exposes $\mathcal{R}$ to obtain a decryption key that can be used to decrypt a challenge ciphertext sent by $\mathcal{I}$. We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_5}(\mathcal{D}_f) = 1$.

Scheme with non-unique signatures.

Let SCh_6 be a variant of our scheme using a digital signature scheme that does not have unique signatures. For concreteness, assume that $\sigma \,\|\, sk$ is a valid signature whenever σ is. Then consider adversary $\mathcal{D}_g$. It exposes $\mathcal{I}$ and has $\mathcal{I}$ send a challenge ciphertext. Then it modifies the ciphertext by changing the signature and forwards this modified ciphertext on to $\mathcal{R}$. The adversary is given back the true decryption of this ciphertext (because it was changed) which trivially reveals the secret bit of the game (here it is important that the signature is not part of the label used for encryption/decryption). We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_6}(\mathcal{D}_g) = 1$.

Scheme with symmetric primitives.

Let SCh_7 be a variant of our scheme that uses a MAC instead of a digital signature scheme (e.g. $vk = sk$ always, and vk is presumably no longer sent in the clear with the ciphertext). Consider adversary $\mathcal{D}_h$. It simply exposes $\mathcal{I}$ and then uses $\mathcal{I}$'s vk to send a message to $\mathcal{I}$. This trivially allows it to determine the secret bit. Here we used that PKE will decrypt any ciphertext to a non-$\perp$ value. We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_7}(\mathcal{D}_h) = 1$.

Similarly let SCh_8 be a variant of our scheme that uses symmetric encryption instead of public-key encryption (e.g. $ek = dk$ always, and ek is presumably no longer sent in the clear with the ciphertext). Adversary $\mathcal{D}_i$ exposes user $\mathcal{I}$ and then uses the corresponding ek to decrypt a challenge message encrypted by $\mathcal{I}$. We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_8}(\mathcal{D}_i) = 1$.

Stated broadly, a scheme that relies on symmetric primitives will not be secure because a user will know sufficient information to send a ciphertext that they would themselves accept or to read a message that they sent to the other user. Our security notion requires that this is not possible.

2.5.2 Security proof

The following theorem bounds the advantage of an adversary breaking the AEAC security of SCh using the advantages of adversaries against the CR security of H, the UFEXP and UNIQ security of DS, the INDEXP security of PKE, and the min-entropy of DS and PKE.

Theorem 9. *Let* DS *be a key-updatable digital signature scheme,* PKE *be a key-updatable public-key encryption scheme, and* H *be a family of functions. Let* $\mathsf{SCh} = \mathsf{SCH}[\mathsf{DS}, \mathsf{PKE}, \mathsf{H}]$. *Let* $\mathcal{D}$ *be an adversary making at most* q_{SEND} *queries to its* SEND *oracle,* q_{RECV} *queries to its* RECV *oracle, and* q_{EXP} *queries to its* EXP *oracle. Then we can build adversaries* $\mathcal{A}_{\mathsf{H}}$, $\mathcal{A}_{\mathsf{DS}}$, $\mathcal{B}_{\mathsf{DS}}$, *and* $\mathcal{A}_{\mathsf{PKE}}$ *such that*

$$\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}}(\mathcal{D}) \leq 2 \cdot (q_{\text{SEND}} \cdot 2^{-\mu} + \mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathcal{A}_{\mathsf{H}}) + \mathsf{Adv}^{\mathsf{ufexp}}_{\mathsf{DS}}(\mathcal{A}_{\mathsf{DS}})$$
$$+ \mathsf{Adv}^{\mathsf{uniq}}_{\mathsf{DS}}(\mathcal{B}_{\mathsf{DS}})) + \mathsf{Adv}^{\mathsf{indexp}}_{\mathsf{PKE}}(\mathcal{A}_{\mathsf{PKE}})$$

where $\mu = \mathrm{H}_\infty(\mathsf{DS.Kg}) + \mathrm{H}_\infty(\mathsf{PKE.Kg}) + \mathrm{H}_\infty(\mathsf{PKE.Enc})$. *Adversary* $\mathcal{A}_{\mathsf{DS}}$ *makes at most* $q_{\text{SEND}} + 2$ *queries to its* NEWUSER *oracle,* q_{SEND} *queries to its* SIGN *oracle, and* q_{EXP} *queries to its* EXP *oracle. Adversary* $\mathcal{B}_{\mathsf{DS}}$ *makes at most* $q_{\text{SEND}} + 2$ *queries to its* NEWUSER *oracle. Adversary* $\mathcal{A}_{\mathsf{PKE}}$ *makes at most* $q_{\text{SEND}} + 2$ *queries to its* NEWUSER *oracle,* $q_{\text{SEND}} * (q_{\text{RECV}} + 1)$ *queries to its* UPDEK *oracle,* $(q_{\text{SEND}} + 1) * q_{\text{RECV}}$ *queries to its* UPDDK *oracle,* q_{SEND} *queries to its* ENC *oracle,* q_{RECV} *queries to its* DEC *oracle, and* $q_{\text{SEND}} + 2$ *queries to its* EXP *oracle. Adversaries* $\mathcal{A}_{\mathsf{H}}$, $\mathcal{A}_{\mathsf{DS}}$, $\mathcal{B}_{\mathsf{DS}}$, *and* $\mathcal{A}_{\mathsf{PKE}}$ *all have runtime about that of* $\mathcal{D}$.

The proof broadly consists of two stages. The first stage of the proof (consisting of three game transitions) argues that the adversary will not be able to forge a ciphertext to an unrestricted user except by exposing the other user. This argument is justified by a reduction to an adversary $\mathcal{A}_{\mathsf{DS}}$ against the security of the digital signature scheme. However, care must be taken in this reduction to ensure that $\mathcal{D}$ cannot induce behavior in $\mathcal{A}_{\mathsf{DS}}$ that would result in $\mathcal{A}_{\mathsf{DS}}$ cheating in the digital signature game. Addressing this possibility involves arguing that $\mathcal{D}$ cannot predict

any output of SEND (from whence the min-entropy term in the bound arises) and that it cannot find any collisions in the hash function H.

Once this stage is complete the output of RECV no longer depends on the secret bit b, so we move to using the security of PKE to argue that $\mathcal{D}$ cannot use SEND to learn the value of the secret bit. This is the second stage of the proof. But prior to this reduction we have to make one last argument using the security of DS. Specifically we show that, given a ciphertext (σ, v), the adversary will not be able to find a new signature σ' such that (σ', v) will be accepted by the receiver (otherwise since $\sigma \neq \sigma'$, oracle RECV would return the true decryption of this ciphertext which would be the same as the decryption of the original ciphertext and thus allow a trivial attack). Having done this, the reduction to the security of PKE is follows.

Theorem 9. For the proof we use games G_0, G_1, G_2, G_3 defined in Fig. 2.13, games G_4, G_5 defined in Fig. 2.16, and the adversaries defined in Fig. 2.14, 2.15, 2.17, 2.18.

The theorem follows immediately from the following seven claims. The first four claims correspond to the first stage of the proof as discussed in Section 2.5.2. The rest correspond to the second stage. For compactness we will let $q = q_{\text{SEND}}$ throughout the proof.

1. $\Pr[G_0] = \Pr[\text{AEAC}_{\mathsf{SCh}}^{\mathcal{D}}]$
2. $\Pr[G_0] - \Pr[G_1] \leq q \cdot 2^{-\mu}$
3. $\Pr[G_1] - \Pr[G_2] \leq \mathsf{Adv}_{\mathsf{H}}^{\mathsf{cr}}(\mathcal{A}_{\mathsf{H}})$
4. $\Pr[G_2] - \Pr[G_3] \leq \mathsf{Adv}_{\mathsf{DS}}^{\mathsf{ufexp}}(\mathcal{A}_{\mathsf{DS}})$
5. $\Pr[G_3] = \Pr[G_4]$
6. $\Pr[G_4] - \Pr[G_5] \leq \mathsf{Adv}_{\mathsf{DS}}^{\mathsf{uniq}}(\mathcal{B}_{\mathsf{DS}})$
7. $\Pr[G_5] \leq \Pr[\text{INDEXP}_{\mathsf{PKE}}^{\mathcal{A}_{\mathsf{PKE}}}]$

Referring to the adversaries from Section 2.5.1 we can roughly think as follows. Claim 1 and Claim 2 will rule out attacks like $\mathcal{D}_a$, $\mathcal{D}_b$, and $\mathcal{D}_c$. Claim 4 will rule out attacks like $\mathcal{D}_e$ and $\mathcal{D}_h$. Claim 6 will rule out attacks like $\mathcal{D}_g$. Claim 7 will rule out attacks like $\mathcal{D}_d$, $\mathcal{D}_f$, and $\mathcal{D}_i$.

Then the relevant calculation is as follows

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}}(\mathcal{D}) =& 2\Pr[\mathrm{AEAC}^{\mathcal{D}}_{\mathsf{SCh}}] - 1 = 2\Pr[\mathrm{G}_0] - 1 \\
=& 2(\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_1] + \Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_2] + \Pr[\mathrm{G}_2] - \Pr[\mathrm{G}_3] \\
& + \Pr[\mathrm{G}_3] - \Pr[\mathrm{G}_4] + \Pr[\mathrm{G}_4] - \Pr[\mathrm{G}_5] + \Pr[\mathrm{G}_5]) - 1 \\
\leq& 2(q \cdot 2^{-\mu} + \mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathcal{A}_{\mathsf{H}}) + \mathsf{Adv}^{\mathsf{ufexp}}_{\mathsf{DS}}(\mathcal{A}_{\mathsf{DS}}) + 0 + \mathsf{Adv}^{\mathsf{uniq}}_{\mathsf{DS}}(\mathcal{B}_{\mathsf{DS}})) \\
& + 2\Pr[\mathrm{INDEXP}^{\mathcal{A}_{\mathsf{PKE}}}_{\mathsf{PKE}}] - 1 \\
=& 2(q \cdot 2^{-\mu} + \mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathcal{A}_{\mathsf{H}}) + \mathsf{Adv}^{\mathsf{ufexp}}_{\mathsf{DS}}(\mathcal{A}_{\mathsf{DS}}) + \mathsf{Adv}^{\mathsf{uniq}}_{\mathsf{DS}}(\mathcal{B}_{\mathsf{DS}})) + \mathsf{Adv}^{\mathsf{indexp}}_{\mathsf{PKE}}(\mathcal{A}_{\mathsf{PKE}}).
\end{aligned}$$

To prove the first four claims we need to consider the sequence of games shown in Fig. 2.13. Lines of code annotated with comments are only in the specified games, all other code is common to all game. We used boxes to emphasize the lines of code annotated with comments. We use highlighting to emphasize the areas of new code.

The first three game transitions (from G_0 to G_3) will show that any ciphertext sent to u by the adversary can be rejected unless either it equals a ciphertext created by $\overline{u}$, u is already restricted, or the adversary has exposed $\overline{u}$ to steal the corresponding signing key. As part of this the transitions from G_0 to G_2 will show roughly that setting a restricted flag results in the two users having differing transcripts.

Claim 1, $\Pr[\mathrm{G}_0] = \Pr[\mathrm{AEAC}^{\mathcal{D}}_{\mathsf{SCh}}]$**.**

Game G_0 was created by hardcoding the code of SCh into $\mathrm{AEAC}^{\mathcal{D}}$, flipping the position of the two if statements in EXP, and then adding some variables to help us transition to future games, which we will describe momentarily. Rather than storing the state of each user as a tuple we store the components of the state in separate variables with an underscript to denote which user's state they are from. None of these variables have any effect on the input-output behavior of G_0 so the first claim, $\Pr[\mathrm{G}_0] = \Pr[\mathrm{AEAC}^{\mathcal{D}}_{\mathsf{SCh}}]$ is immediate.

```
Games G_0, G_1, G_2, G_3
1:  b ←$ {0,1} ; r_I^{rest} ← r_R^{rest} ← ∞
2:  s_I ← r_I ← s_R ← r_R ← 0
3:  restricted_I ← restricted_R ← false
4:  forge_I[0...∞] ← "nontrivial"
5:  forge_R[0...∞] ← "nontrivial"
6:  X_I ← X_R ← 0 ; hk ←$ H.Kg
7:  (sk_I, vk_R) ←$ DS.Kg ; (sk_R, vk_I) ←$ DS.Kg
8:  (ek_I, dk_R[0]) ←$ PKE.Kg ; (ek_R, dk_I[0]) ←$ PKE.Kg
9:  τ_{r,I}[0] ← τ_{r,R}[0] ← τ_{s,I}[0] ← τ_{s,R}[0] ← ε
10: r_I^{ack} ← r_R^{ack} ← r_I^{sent} ← r_R^{sent} ← 0
11: unchanged_I ← unchanged_R ← true
12: (z_I, z_R) ←$ (Ch.SendRS)^2
13: (η_I, η_R) ←$ (Ch.RecvRS)^2
14: b' ←$ D^{Send,Recv,Exp} ; Return (b' = b)

Send(u, m_0, m_1, ad)
15: If nextop ∉ {(u, "sent"), ⊥} then return ⊥
16: If |m_0| ≠ |m_1| then return ⊥
17: t_1 ← (r_u < X_u) ; t_2 ← restricted_u
18: t_3 ← (ch_u[s_u + 1] = "forbidden")
19: If (t_1 or t_2 or t_3) and m_0 ≠ m_1 then return ⊥
20: (z_1, z_2, z_3, z_4) ← z_u ; s_u ← s_u + 1
21: (sk, vk) ← DS.Kg(z_1)
22: (ek, dk_u[s_u]) ← PKE.Kg(z_2)
23: ℓ ← (s_u, r_u, ad, vk, ek, τ_{r,u}[r_u], τ_{s,u}[s_u − 1])
24: Δ_e ← τ_{s,u}[r_u^{ack} + 1, ..., s_u − 1]
25: (ek', c') ← PKE.Enc(ek_u, ℓ, m_b, Δ_e; z_3)
26: v ← (c', ℓ)
27: σ ← DS.Sign(sk_u, v; z_4) ; c ← (σ, v)
28: τ_{s,u}[s_u] ← H.Ev(hk, c) ; sk_u ← sk
29: r_u^{sent} ← r_u ; nextop ← ⊥ ; z_u ←$ Ch.SendRS
30: sctable_{ū}[s_u] ← (c, ad)
31: If unchanged_u and τ_{s,u}[s_u] = τ_{r,ū}[s_u] then
32:     If rctable_{ū}[s_u] = (c, ad) then
33:         bad_pred ← true
34:         abort(false)   / G_1, G_2, G_3
35:     Else
36:         bad_coll ← true
37:         abort(false)   / G_2, G_3
38: If not restricted_u then ctable_{ū}[s_u] ← (c, ad)
39: If m_0 ≠ m_1 then ch_u[s_u] ← "done"
40: Return c

Recv(u, c, ad)
41: If nextop ∉ {(u, "recv"), ⊥} then return ⊥
42: (η_1, η_2) ← η_u ; nextop ← ⊥ ; η_u ←$ Ch.RecvRS
43: (σ, v) ← c ; (c', ℓ) ← v
44: (s', r', ad', vk', ek', τ'_r, τ'_s) ← ℓ
45: Δ ← τ_{s,u}[r_u^{ack} + 1, ..., r']
46: (vk'', t) ← DS.Vrfy(vk_u, σ, v, Δ)
47: t_4 ← ((s', τ'_r) ≠ (r_u + 1, τ_{s,u}[r']))
48: t_5 ← ((τ'_s, ad') ≠ (τ_{r,u}[r_u], ad)))
49: If not t or t_4 or t_5 then return ⊥
50: t_a ← (forge_u[r_u + 1] ≠ "trivial")
51: t_b ← ((c, ad) ≠ ctable_u[r_u + 1])
52: If unchanged_u and t_a and t_b then
53:     bad_forge ← true
54:     abort(false)   / G_3
55: r_u ← r_u + 1 ; r_u^{ack} ← r'
56: m ← PKE.Dec(dk_u[r_u^{ack}], ℓ, c')
57: τ_{s,u}[0, ..., r_u^{ack} − 1] ← ⊥
58: dk_u[0, ..., r_u^{ack} − 1] ← ⊥
59: τ_{r,u}[r_u] ← H.Ev(hk, c)
60: sk_u ← DS.UpdSk(sk_u, τ_{r,u}[r_u]; η_1)
61: vk_u ← vk' ; ek_u ← ek'
62: For i ∈ [r_u^{ack}, s_u] do
63:     dk_u[i] ← PKE.UpdDk(dk_u[i], τ_{r,u}[r_u]; η_2^i)
64: rctable_u[r_u] ← (c, ad)
65: If t_b then unchanged_u ← false
66: If forge_u[r_u] = "trivial" and t_b then
67:     restricted_u ← true ; r_u^{rest} ← min{r_u, r_u^{rest}}
68: If unchanged_{ū} and τ_{r,u}[r_u^{rest}] = τ_{s,ū}[r_u^{rest}] ≠ ⊥ then
69:         bad_coll ← true
70:         abort(false)   / G_2, G_3
71: If restricted_u or (b = 0 and t_b) then
72:     Return m
73: Return ⊥

Exp(u, rand)
74: If nextop ≠ ⊥ then return ⊥
75: If not restricted_u and ∃i ∈ (r_u, s_ū] s.t. ch_ū[i] = "done" then return ⊥
76: k ← (s_u, r_u, sk_u, vk_u, ek_u, dk_u, hk, τ_{r,u}[r_u], τ_{s,u})
77: If restricted_u then return (k, z_u, η_u)
78: forge_ū[s_u + 1] ← "trivial" ; (z, η) ← (ε, ε) ; X_ū ← s_u + 1
79: If rand = "send" then
80:     nextop ← (u, "send") ; z ← z_u
81:     X_ū ← s_u + 2 ; forge_ū[s_u + 2] ← "trivial"
82:     ch_u[s_u + 1] ← "forbidden"
83: Else if rand = "recv" then
84:     nextop ← (u, "recv") ; η ← η_u
85: Return (k, z, η)
```

Figure 2.13. Games G_1, G_2, and G_3 for security proof. Lines commented with the names of games are only included in those games. Highlighting indicates new code.

Descriptions of games G_1, G_2, and G_3.

Variables t_1 through t_5 and $\vec{\Delta}_e$ are temporary variables which were added to make the code more compact. Their uses are not highlighted.

Variables r_u^{sent} and r_u^{rest} store indices corresponding to information about the ciphertexts that a user has received. In r_u^{sent} we store the last value of r_u that was included in a ciphertext sent by u (i.e. the index of the last ciphertext that u has told $\overline{u}$ they have received). This variable has no effect on the code, but will be useful for reasoning about its behavior. In r_u^{rest} we store the value held by r_u when $\mathsf{restricted}_u$ was first set or ∞ if this has not occurred.

The flag $\mathsf{unchanged}_u$ will be initialized to true and will be set false the first time that the sequence of ciphertexts received by u differs from the sequence of ciphertext sent by $\overline{u}$ while unrestricted (i.e. the first time its view is "changed" from the correct view output by $\overline{u}$). Once $\overline{u}$ is restricted **ctable** would not be set on line 38, so a the corresponding t_b in $\textsc{Recv}$ would have to be false. This flag is not the exact opposite of $\mathsf{restricted}_u$ because $\mathsf{restricted}_u$ only gets set to true if the changed ciphertext was sent after an exposure. However, it is useful to note that $\mathsf{restricted}_u$ will always be false when $\mathsf{unchanged}_u$ is true.

Table $\mathbf{rctable}_u$ stores the sequence of ciphertexts that have been received and accepted by u. Table $\mathbf{sctable}_u$ stores the sequence of ciphertexts that have been sent by $\overline{u}$. This differs from the existing table $\mathbf{ctable}_u$ because it continues to store these values even after $\mathsf{restricted}_{\overline{u}}$ has been set.

On every call to $\textsc{Recv}$, we use $\vec{\Delta}$ as a temporary variable to store the sequence of transcripts used to update the digital signature verification key.

We seek to argue that before the first time u's view is changed, the adversary should not be able to create any (c, ad) pairs that will be accepted by u unless the pair was output by $\overline{u}$ or the adversary has done an appropriate exposure of $\overline{u}$. This belief about (c, ad) pairs provided by the adversary that should be rejected is encoded in the if statement in $\textsc{Recv}$ on line 52 which checks if $\mathsf{unchanged}_u$, $\mathbf{forge}_u[r_u+1] \neq$ "trivial", and $(c, ad) \neq \mathbf{ctable}_u[r_u+1]$. Once we reach G_3 we will simply abort if this evaluates to true.

The first two game transitions (to games G_1 and G_2) are used to rule out the possibility that the other user is restricted, but has receiving transcripts which match the sending transcripts of the current user. This possibility is captured by the if statements on lines 31 and 68. Were this possible then forgeries would be possible because after the other user is restricted the adversary can have their secrets exposed without setting the appropriate entry of $\mathbf{forge}_{\overline{u}}$ to "trivial". This will be used again at the end of the proof because if this were possible it would also lead to an attack against the security provide by the encryption scheme.

Claim 2, $\Pr[G_0] - \Pr[G_1] \leq q \cdot 2^{-\mu}$**.**

To start this argument consider the flag bad_{pred}. Games G_0 and G_1 are identical until bad_{pred}. So from the fundamental lemma of game playing [16] we have $\Pr[G_0] - \Pr[G_1] \leq \Pr[G_0 \text{ sets } \mathsf{bad}_{pred}]$. To establish the second claim we need to show that $\Pr[G_0 \text{ sets } \mathsf{bad}_{pred}] \leq q \cdot 2^{-\mu}$. This will hold because an adversary can only cause bad_{pred} to be set by predicting the output of Ch.Send before it is generated (which requires predicting the output of DS.Kg, PKE.Kg, and PKE.Enc).

This flag gets set in SEND when $\mathsf{unchanged}_u$, $\tau_{s,u}[s_u] = \tau_{r,\overline{u}}[s_u]$, and $\mathbf{rctable}_{\overline{u}}[s_u] = (c, ad)$ are all true. Note that because $\mathsf{unchanged}_u$ is true, it must hold that $\mathsf{restricted}_u$ is false. Thus an exposure of the randomness used by the SEND operation would set nextop to $(u, \text{"send"})$ and could thus only happen immediately before the SEND operation (and after the RECV operation during which $\mathbf{rctable}_{\overline{u}}[s_u]$ was set). This means the adversary must have predicted the output of SEND before making the call. By the properties of min-entropy, the probability this happens from any individual RECV call is at most $2^{-\mu}$ and then by a union bound we get the desired second claim.

Claim 3, $\Pr[G_1] - \Pr[G_2] \leq \mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathcal{A}_{\mathsf{H}})$**.**

For the third claim note that games G_1 and G_2 are identical until bad_{coll} so $\Pr[G_1] - \Pr[G_2] \leq \Pr[G_1 \text{ sets } \mathsf{bad}_{coll}]$.

When this flag is set in SEND it must hold that $\mathbf{sctable}_{\overline{u}}[s_u] \neq \mathbf{rctable}_{\overline{u}}[s_u]$ yet $\tau_{s,u}[s_u] =$

Adversary $\mathcal{A}_{\mathsf{H}}(hk)$

86: $b \leftarrow\!\!\$\, \{0,1\}\,;\; r_{\mathcal{I}}^{rest} \leftarrow r_{\mathcal{R}}^{rest} \leftarrow \infty\,;\; s_{\mathcal{I}} \leftarrow r_{\mathcal{I}} \leftarrow s_{\mathcal{R}} \leftarrow r_{\mathcal{R}} \leftarrow 0\,;\; \mathsf{restricted}_{\mathcal{I}} \leftarrow \mathsf{restricted}_{\mathcal{R}} \leftarrow \mathsf{false}$

87: $\mathbf{forge}_{\mathcal{I}}[0\ldots\infty] \leftarrow \mathbf{forge}_{\mathcal{R}}[0\ldots\infty] \leftarrow$ "nontrivial"; $\mathcal{X}_{\mathcal{I}} \leftarrow \mathcal{X}_{\mathcal{R}} \leftarrow 0$

88: $(sk_{\mathcal{I}}, vk_{\mathcal{R}}) \leftarrow\!\!\$\, \mathsf{DS.Kg}\,;\; (sk_{\mathcal{R}}, vk_{\mathcal{I}}) \leftarrow\!\!\$\, \mathsf{DS.Kg}$

89: $(ek_{\mathcal{I}}, \mathbf{dk}_{\mathcal{R}}[0]) \leftarrow\!\!\$\, \mathsf{PKE.Kg}\,;\; (ek_{\mathcal{R}}, \mathbf{dk}_{\mathcal{I}}[0]) \leftarrow\!\!\$\, \mathsf{PKE.Kg}$

90: $hk \leftarrow hk\,;\; \tau_{r,\mathcal{I}}[0] \leftarrow \tau_{r,\mathcal{R}}[0] \leftarrow \tau_{s,\mathcal{I}}[0] \leftarrow \tau_{s,\mathcal{R}}[0] \leftarrow \varepsilon$

91: $\mathsf{unchanged}_{\mathcal{I}} \leftarrow \mathsf{unchanged}_{\mathcal{R}} \leftarrow \mathsf{true}\,;\; r_{\mathcal{I}}^{ack} \leftarrow r_{\mathcal{R}}^{ack} \leftarrow r_{\mathcal{I}}^{sent} \leftarrow r_{\mathcal{R}}^{sent} \leftarrow 0$

92: $(z_{\mathcal{I}}, z_{\mathcal{R}}) \leftarrow\!\!\$\, (\mathsf{Ch.SendRS})^2\,;\; (\eta_{\mathcal{I}}, \eta_{\mathcal{R}}) \leftarrow\!\!\$\, (\mathsf{Ch.RecvRS})^2$

93: $\mathcal{D}^{\text{SendSim},\text{RecvSim},\text{ExpSim}}\,;$ Return $(\varepsilon, \varepsilon)$

$\text{SendSim}(u, m_0, m_1, ad)$

If $\mathsf{nextop} \notin \{(u, \text{"sent"}), \bot\}$ then return $\bot$

94: If $|m_0| \neq |m_1|$ then return $\bot$

95: If ($r_u < \mathcal{X}_u$ or $\mathsf{restricted}_u$ or $\mathbf{ch}_u[s_u+1] =$ "forbidden") and $m_0 \neq m_1$ then return $\bot$

96: $(z_1, z_2, z_3, z_4) \leftarrow z_u\,;\; s_u \leftarrow s_u + 1\,;\; (sk, vk) \leftarrow \mathsf{DS.Kg}(z_1)\,;\; (ek, \mathbf{dk}_u[s_u]) \leftarrow \mathsf{PKE.Kg}(z_2)$

97: $\ell \leftarrow (s_u, r_u, ad, vk, ek, \tau_{r,u}[r_u], \tau_{s,u}[s_u - 1])\,;\; \vec{\Delta}_e \leftarrow \tau_{s,u}[r_u^{ack}+1, \ldots, s_u - 1]$

98: $(ek', c') \leftarrow \mathsf{PKE.Enc}(ek_u, \ell, m_b, \vec{\Delta}_e; z_3)\,;\; v \leftarrow (c', \ell)\,;\; \sigma \leftarrow \mathsf{DS.Sign}(sk_u, v; z_4)\,;\; c \leftarrow (\sigma, v)$

99: $\tau_{s,u}[s_u] \leftarrow \mathsf{H.Ev}(hk, c)\,;\; sk_u \leftarrow sk\,;\; r_u^{sent} \leftarrow r_u\,;\; \mathsf{nextop} \leftarrow \bot\,;\; z_u \leftarrow\!\!\$\, \mathsf{Ch.SendRS}$

100: $\quad \mathbf{sctable}_{\overline{u}}[s_u] \leftarrow (c, ad)$

101: $\quad$ If $\mathsf{unchanged}_u$ and $\tau_{s,u}[s_u] = \tau_{r,\overline{u}}[s_u]$ then

102: $\quad\quad$ If $\mathbf{rctable}_{\overline{u}}[s_u] = (c, ad)$ then $\mathbf{abort}(\varepsilon, \varepsilon)$

103: $\quad\quad$ Else $(c, ad) \leftarrow \mathbf{sctable}_{\overline{u}}[s_u]\,; (c', ad') \leftarrow \mathbf{rctable}_{\overline{u}}[s_u]\,;\; \mathbf{abort}(c, c')$

104: $\quad$ If not $\mathsf{restricted}_u$ then $\mathbf{ctable}_{\overline{u}}[s_u] \leftarrow (c, ad)$

105: $\quad$ If $m_0 \neq m_1$ then $\mathbf{ch}_u[s_u] \leftarrow$ "done"

106: $\quad$ Return c

$\text{RecvSim}(u, c, ad)$

107: If $\mathsf{nextop} \notin \{(u, \text{"recv"}), \bot\}$ then return $\bot$

108: $(\eta_1, \eta_2) \leftarrow \eta_u\,;\; \mathsf{nextop} \leftarrow \bot\,;\; \eta_u \leftarrow\!\!\$\, \mathsf{Ch.RecvRS}\,;\; (\sigma, v) \leftarrow c\,;\; (c', \ell) \leftarrow v$

109: $(s', r', ad', vk', ek', \tau_r', \tau_s') \leftarrow \ell\,;\; \vec{\Delta} \leftarrow \tau_{s,u}[r_u^{ack}+1, \ldots, r']\,;\; (vk'', t) \leftarrow \mathsf{DS.Vrfy}(vk_u, \sigma, v, \vec{\Delta})$

110: If not t or $s' \neq r_u + 1$ or $\tau_r' \neq \tau_{s,u}[r']$ or $\tau_s' \neq \tau_{r,u}[r_u]$ or $ad' \neq ad$ then return $\bot$

111: $t_b \leftarrow ((c, ad) \neq \mathbf{ctable}_u[r_u + 1])\,;\; r_u \leftarrow r_u + 1\,;\; r_u^{ack} \leftarrow r'\,;\; m \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}_u[r_u^{ack}], \ell, c')$

112: $\tau_{s,u}[0, \ldots, r_u^{ack} - 1] \leftarrow \bot\,;\; \mathbf{dk}_u[0, \ldots, r_u^{ack} - 1] \leftarrow \bot\,;\; \tau_{r,u}[r_u] \leftarrow \mathsf{H.Ev}(hk, c)$

113: $sk_u \leftarrow \mathsf{DS.UpdSk}(sk_u, \tau_{r,u}[r_u]; \eta_1)\,;\; vk_u \leftarrow vk'\,;\; ek_u \leftarrow ek'\,;\; \mathbf{rctable}_u[r_u] \leftarrow (c, ad)$

114: For $i \in [r_u^{ack}, s_u]$ do $\mathbf{dk}_u[i] \leftarrow \mathsf{PKE.UpdDk}(\mathbf{dk}_u[i], \tau_{r,u}[r_u]; \eta_2^i)$

115: If t_b then $\mathsf{unchanged}_u \leftarrow \mathsf{false}$

116: If $\mathbf{forge}_u[r_u] =$ "trivial" and t_b then $\mathsf{restricted}_u \leftarrow \mathsf{true}\,;\; r_u^{rest} \leftarrow \min\{r_u, r_u^{rest}\}$

117: If $\mathsf{unchanged}_{\overline{u}}$ and $\tau_{r,u}[r_u^{rest}] = \tau_{s,\overline{u}}[r_u^{rest}] \neq \bot$ then

118: $\quad (c', ad') \leftarrow \mathbf{rctable}_u[r_u^{rest}]\,;\; (c, ad) \leftarrow \mathbf{sctable}_u[r_u^{rest}]\,;\; \mathbf{abort}(c, c')$

119: If $\mathsf{restricted}_u$ or ($b = 0$ and t_b) then return m

120: Return $\bot$

$\text{ExpSim}(u, \mathsf{rand})$

// Identical to Exp in $\mathrm{G}_1, \mathrm{G}_2$

Figure 2.14. Adversary $\mathcal{A}_{\mathsf{H}}$ against collision resistance of H. Highlighting indicates the changes from $\mathrm{G}_1, \mathrm{G}_2$.

$\tau_{r,\overline{u}}[s_u]$. So the corresponding ciphertexts must have formed a collision in H. (Note here that ciphertexts include the associated date, so $\mathbf{sctable}_{\overline{u}}[r_{\overline{u}}^{rest}] \neq \mathbf{rctable}_{\overline{u}}[r_{\overline{u}}^{rest}]$ implies the corresponding ciphertexts are distinct.)

When want to argue that when this flag is set in Recv it must hold that $\mathbf{rctable}_u[r_u^{rest}] \neq \mathbf{sctable}_u[r_u^{rest}]$ even though $\tau_{r,u}[r_u^{rest}] = \tau_{s,\overline{u}}[r_u^{rest}] \neq \bot$. Not that because $\mathsf{unchanged}_{\overline{u}}$ is true, $\mathsf{restricted}_{\overline{u}}$ must be false. If $\tau_{r,u}[r_u^{rest}]$ was set to a non-$\bot$ value first, then bad_{coll} would have

already been set true in SEND when $\tau_{s,\overline{u}}[r_u^{rest}]$ was set. So suppose $\tau_{s,\overline{u}}[r_u^{rest}]$ was set first and consider the first time it was set. The variable t_b was true at this time so we had $\mathbf{rctable}_u[r_u^{rest}] = (c, ad) \neq \mathbf{ctable}_u[r_u^{rest}] = \mathbf{sctable}_u[r_u^{rest}]$. (The last equality held because $\mathsf{restricted}_{\overline{u}}$ is still false.) The same reasoning as the previous paragraph implies that the corresponding ciphertexts must have formed a collision in H.

The adversary $\mathcal{A}_{\mathsf{H}}$ shown in Fig. 2.14 takes advantage of this to attack the collision-resistance of H. It simulates the view of $\mathcal{D}$ and then (on line 103 or line 118) aborts and returns the collision when bad_{coll} would be set; otherwise it simply gives up and returns $(\varepsilon, \varepsilon)$. The highlighted code shows where the code of $\mathcal{A}_{\mathsf{H}}$ differs from that of G_1 and G_2. The code $hk \leftarrow hk$ is included just to make explicit that it uses the key to the hash function it was provided as input rather than sampling a new key for itself. That $\mathcal{A}_{\mathsf{H}}$ perfectly simulates the view of $\mathcal{D}$ in G_1 or G_2 until bad_{coll} is clear. From our above analysis we then have that $\mathcal{A}_{\mathsf{H}}$ succeeds in CR whenever G_1 would set bad_{coll} so $\Pr[G_1 \text{ sets } \mathsf{bad}_{coll}] \leq \Pr[\mathrm{CR}_{\mathsf{H}}^{\mathcal{A}_{\mathsf{H}}}] = \mathsf{Adv}_{\mathsf{H}}^{\mathsf{cr}}(\mathcal{A}_{\mathsf{H}})$ as desired.

Claim 4, $\Pr[G_2] - \Pr[G_3] \leq \mathsf{Adv}_{\mathsf{DS}}^{\mathsf{ufexp}}(\mathcal{A}_{\mathsf{DS}})$**.**

Finally we can give our reduction to the security of DS to complete this stage of the proof. Game G_2 and G_3 are identical until bad_{forge}, so $\Pr[G_2] - \Pr[G_3] \leq \Pr[G_2 \text{ sets } \mathsf{bad}_{forge}]$. The adversary $\mathcal{A}_{\mathsf{DS}}$ shown in Fig. 2.15 against the security of DS uses its oracles to simulate the view of $\mathcal{D}$ and then (on line 174) aborts and returns the forgery if bad_{forge} would ever be set. The highlighting shows where the code of $\mathcal{A}_{\mathsf{DS}}$ differs from that of games G_2 and G_3. Below we verify that $\mathcal{A}_{\mathsf{DS}}$ correctly simulates the view of $\mathcal{D}$ in these games (until the time that it aborts).

Now consider a call to $\mathrm{RECVSIM}(u, \cdot, \cdot)$ by $\mathcal{D}$ that results in $\mathcal{A}_{\mathsf{DS}}$ aborting. Note that $\mathcal{A}_{\mathsf{DS}}$ uses user identifiers of the form $\Lambda = (s, v)$ for the key created by user v with the s-th ciphertext it sends. We seek to argue that $\mathcal{A}_{\mathsf{DS}}$ wins UFEXP whenever it aborts, so we need to show that (1) $vk[(r_u, \overline{u})] \neq \perp$, (2) $\mathsf{win} = \mathsf{true}$, and (3) $\mathsf{cheated} = \mathsf{false}$. Then because $\mathcal{A}_{\mathsf{DS}}$ aborts whenever bad_{forge} would be set we get $\Pr[G_2 \text{ sets } \mathsf{bad}_{forge}] \leq \Pr[\mathrm{UFEXP}_{\mathsf{DS}}^{\mathcal{A}_{\mathsf{DS}}}] = \mathsf{Adv}_{\mathsf{DS}}^{\mathsf{ufexp}}(\mathcal{A}_{\mathsf{DS}})$ as desired.

Adversary $\mathcal{A}_{\mathsf{DS}}^{\textsc{Upd},\textsc{Sign},\textsc{Exp}}$

121: $b \leftarrow\$ \{0,1\}$; $r_{\mathcal{I}}^{rest} \leftarrow r_{\mathcal{R}}^{rest} \leftarrow \infty$
122: $s_{\mathcal{I}} \leftarrow r_{\mathcal{I}} \leftarrow s_{\mathcal{R}} \leftarrow r_{\mathcal{R}} \leftarrow 0$
123: $\mathsf{restricted}_{\mathcal{I}} \leftarrow \mathsf{restricted}_{\mathcal{R}} \leftarrow \mathsf{false}$
124: $\mathbf{forge}_{\mathcal{I}}[0\ldots\infty] \leftarrow$ "nontrivial"
125: $\mathbf{forge}_{\mathcal{R}}[0\ldots\infty] \leftarrow$ "nontrivial"
126: $\mathcal{X}_{\mathcal{I}} \leftarrow \mathcal{X}_{\mathcal{R}} \leftarrow 0$; $hk \leftarrow\$ \mathsf{H.Kg}$
127: $vk_{\mathcal{R}} \leftarrow \textsc{NewUser}((s_{\mathcal{I}},\mathcal{I}))$; $sk_{\mathcal{I}} \leftarrow \bot$
128: $vk_{\mathcal{I}} \leftarrow \textsc{NewUser}((s_{\mathcal{R}},\mathcal{R}))$; $sk_{\mathcal{R}} \leftarrow \bot$
129: $(ek_{\mathcal{I}},\mathbf{dk}_{\mathcal{R}}[0]) \leftarrow\$ \mathsf{PKE.Kg}$; $(ek_{\mathcal{R}},\mathbf{dk}_{\mathcal{I}}[0]) \leftarrow\$ \mathsf{PKE.Kg}$
130: $\tau_{r,\mathcal{I}}[0] \leftarrow \tau_{r,\mathcal{R}}[0] \leftarrow \tau_{s,\mathcal{I}}[0] \leftarrow \tau_{s,\mathcal{R}}[0] \leftarrow \varepsilon$
131: $\mathsf{unchanged}_{\mathcal{I}} \leftarrow \mathsf{unchanged}_{\mathcal{R}} \leftarrow \mathsf{true}$
132: $r_{\mathcal{I}}^{ack} \leftarrow r_{\mathcal{R}}^{ack} \leftarrow r_{\mathcal{I}}^{sent} \leftarrow r_{\mathcal{R}}^{sent} \leftarrow 0$
133: $(z_{\mathcal{I}},z_{\mathcal{R}}) \leftarrow\$ (\mathsf{Ch.SendRS})^2$
134: $(\eta_{\mathcal{I}},\eta_{\mathcal{R}}) \leftarrow\$ (\mathsf{Ch.RecvRS})^2$
135: $\mathcal{D}^{\textsc{SendSim},\textsc{RecvSim},\textsc{ExpSim}}$
136: Return $(\varepsilon,\varepsilon,\varepsilon,(\varepsilon))$

$\textsc{SendSim}(u,m_0,m_1,ad)$

137: If $\mathsf{nextop} \notin \{(u,\text{"sent"}),\bot\}$ then return $\bot$
138: If $|m_0| \neq |m_1|$ then return $\bot$
139: $t_1 \leftarrow (r_u < \mathcal{X}_u)$; $t_2 \leftarrow \mathsf{restricted}_u$
140: $t_3 \leftarrow (\mathbf{ch}_u[s_u+1] =$ "forbidden")
141: If (t_1 or t_2 or t_3) and $m_0 \neq m_1$ then return $\bot$
142: $(z_1,z_2,z_3,z_4) \leftarrow z_u$; $s_u \leftarrow s_u+1$
143: If $\mathsf{nextop} \neq (u,\text{"sent"})$ then
144: $\quad vk \leftarrow \textsc{NewUser}((s_u,u))$; $sk \leftarrow \bot$
145: Else $(sk,vk) \leftarrow \mathsf{DS.Kg}(z_1)$
146: $(ek,\mathbf{dk}_u[s_u]) \leftarrow \mathsf{PKE.Kg}(z_2)$
147: $\ell \leftarrow (s_u,r_u,ad,vk,ek,\tau_{r,u}[r_u],\tau_{s,u}[s_u-1])$
148: $\vec{\Delta}_e \leftarrow \tau_{s,u}[r_u^{ack}+1,\ldots,s_u-1]$
149: $(ek',c') \leftarrow \mathsf{PKE.Enc}(ek_u,\ell,m_b,\vec{\Delta}_e;z_3)$
150: $v \leftarrow (c',\ell)$
151: If $sk_u = \bot$ then $\sigma \leftarrow \textsc{Sign}((s_u-1,u),v)$
152: Else $\sigma \leftarrow \mathsf{DS.Sign}(sk_u,v;z_4)$
153: $c \leftarrow (\sigma,v)$; $\tau_{s,u}[s_u] \leftarrow \mathsf{H.Ev}(hk,c)$; $sk_u \leftarrow sk$
154: $r_u^{sent} \leftarrow r_u$; $\mathsf{nextop} \leftarrow \bot$; $z_u \leftarrow\$ \mathsf{Ch.SendRS}$
155: $\mathbf{sctable}_{\overline{u}}[s_u] \leftarrow (c,ad)$
156: If $\mathsf{unchanged}_u$ and $\tau_{s,u}[s_u] = \tau_{r,\overline{u}}[s_u]$ then
157: $\quad \mathbf{abort}(\varepsilon,\varepsilon,\varepsilon,(\varepsilon))$
158: If not $\mathsf{restricted}_u$ then $\mathbf{ctable}_{\overline{u}}[s_u] \leftarrow (c,ad)$
159: If $m_0 \neq m_1$ then $\mathbf{ch}_u[s_u] \leftarrow$ "done"
160: Return c

$\textsc{RecvSim}(u,c,ad)$

161: If $\mathsf{nextop} \notin \{(u,\text{"recv"}),\bot\}$ then return $\bot$
162: $(\eta_1,\eta_2) \leftarrow \eta_u$; $\mathsf{nextop} \leftarrow \bot$
163: $\eta_u \leftarrow\$ \mathsf{Ch.RecvRS}$
164: $(\sigma,v) \leftarrow c$; $(c',\ell) \leftarrow v$
165: $(s',r',ad',vk',ek',\tau_r',\tau_s') \leftarrow \ell$
166: $\vec{\Delta} \leftarrow \tau_{s,u}[r_u^{ack}+1,\ldots,r']$
167: $(vk'',t) \leftarrow \mathsf{DS.Vrfy}(vk_u,\sigma,v,\vec{\Delta})$
168: $t_4 \leftarrow ((s',\tau_r') \neq (r_u+1,\tau_{s,u}[r']))$
169: $t_5 \leftarrow \big((\tau_s',ad') \neq (\tau_{r,u}[r_u],ad)\big)$
170: If not t or t_4 or t_5 then return $\bot$
171: $t_a \leftarrow (\mathbf{forge}_u[r_u+1] \neq$ "trivial")
172: $t_b \leftarrow ((c,ad) \neq \mathbf{ctable}_u[r_u+1])$
173: If $\mathsf{unchanged}_u$ and t_a and t_b then
174: $\quad \mathbf{abort}((r_u,\overline{u}),\sigma,v,\vec{\Delta})$
175: $r_u \leftarrow r_u+1$; $r_u^{ack} \leftarrow r'$
176: $m \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}_u[r_u^{ack}],\ell,c')$
177: $\tau_{s,u}[0,\ldots,r_u^{ack}-1] \leftarrow \bot$
178: $\mathbf{dk}_u[0,\ldots,r_u^{ack}-1] \leftarrow \bot$
179: $\tau_{r,u}[r_u] \leftarrow \mathsf{H.Ev}(hk,c)$
180: If $sk_u = \bot$ then $\textsc{Upd}((s_u,u),\tau_{r,u}[r_u])$
181: Else $sk_u \leftarrow \mathsf{DS.UpdSk}(sk_u,\tau_{r,u}[r_u];\eta_1)$
182: $vk_u \leftarrow vk'$; $ek_u \leftarrow ek'$
183: For $i \in [r_u^{ack},s_u]$ do
184: $\quad \mathbf{dk}_u[i] \leftarrow \mathsf{PKE.UpdDk}(\mathbf{dk}_u[i],\tau_{r,u}[r_u];\eta_1^i)$
185: $\mathbf{rctable}_u[r_u] \leftarrow (c,ad)$
186: If t_b then $\mathsf{unchanged}_u \leftarrow \mathsf{false}$
187: If $\mathbf{forge}_u[r_u] =$ "trivial" and t_b then
188: $\quad \mathsf{restricted}_u \leftarrow \mathsf{true}$; $r_u^{rest} \leftarrow \min\{r_u,r_u^{rest}\}$
189: If $\mathsf{unchanged}_{\overline{u}}$ and $\tau_{r,u}[r_u^{rest}] = \tau_{s,\overline{u}}[r_u^{rest}] \neq \bot$ then
190: $\quad \mathbf{abort}(\varepsilon,\varepsilon,\varepsilon,(\varepsilon))$
191: If $\mathsf{restricted}_u$ or ($b = 0$ and t_b) then
192: $\quad$ Return m
193: Return $\bot$

$\textsc{ExpSim}(u,\mathsf{rand})$

194: If $\mathsf{nextop} \neq \bot$ then return $\bot$
195: If not $\mathsf{restricted}_u$ and $\exists i \in (r_u,s_{\overline{u}}]$ s.t. $\mathbf{ch}_{\overline{u}}[i] =$ "done" then return $\bot$
196: If $sk_u = \bot$ then $sk_u \leftarrow \textsc{Exp}((s_u,u))$
197: $\vec{k} \leftarrow (s_u,r_u,sk_u,vk_u,ek_u,\mathbf{dk}_u,hk,\tau_{r,u}[r_u],\tau_{s,u})$
198: If $\mathsf{restricted}_u$ then return $(\vec{k},z_u,\eta_u)$
199: $\mathbf{forge}_{\overline{u}}[s_u+1] \leftarrow$ "trivial" ; $(z,\eta) \leftarrow (\varepsilon,\varepsilon)$; $\mathcal{X}_{\overline{u}} \leftarrow s_u+1$
200: If $\mathsf{rand} =$ "send" then
201: $\quad \mathsf{nextop} \leftarrow (u,\text{"send"})$; $z \leftarrow z_u$; $\mathcal{X}_{\overline{u}} \leftarrow s_u+2$; $\mathbf{forge}_{\overline{u}}[s_u+2] \leftarrow$ "trivial"
202: $\quad \mathbf{ch}_u[s_u+1] \leftarrow$ "forbidden"
203: Else if $\mathsf{rand} =$ "recv" then
204: $\quad \mathsf{nextop} \leftarrow (u,\text{"recv"})$; $\eta \leftarrow \eta_u$
205: Return $(\vec{k},z,\eta)$

Figure 2.15. Adversary $\mathcal{A}_{\mathsf{DS}}$ attacking DS. Highlighting indicates changes from $\mathrm{G}_2,\mathrm{G}_3$

Successful aborts by $\mathcal{A}_{\mathsf{DS}}$.

First note that $\mathsf{unchanged}_u$ is true because of line 173, so vk_u must equal the verification key that was part of the ciphertext sent by $\overline{u}$ when $s_{\overline{u}}$ was equal to the current value r_u. This was output by $\textsc{NewUser}((s_{\overline{u}}, \overline{u}))$ on line 144 unless it held that $\mathsf{nextop} = (\overline{u}, \text{"sent"})$. The only way the latter can hold is if nextop was just set in $\textsc{ExpSim}$, but then $\mathbf{forge}_{\overline{u}}[s_{\overline{u}}+1] = \mathbf{forge}_{\overline{u}}[r_u+1]$ would have been set to "trivial" contradicting that t_a holds in $\textsc{RecvSim}$. (Here note that $s_{\overline{u}}$ was incremented at the start of the $\textsc{SendSim}$ call, so $s_{\overline{u}}+2$ during the exposure was equal to the value of $s_{\overline{u}}+1$ while sending.) So vk_u was output by $\textsc{NewUser}((s_{\overline{u}}, \overline{u}))$ and $vk[(r_u, \overline{u})] \neq \bot$ in UFEXP. This is (1).

That $\mathsf{win} = \mathsf{true}$ in UFEXP follows immediately from the above and the fact that t was true in $\textsc{RecvSim}$ (see line 170). This is (2).

We complete this argument by showing (3) that if $\mathsf{cheated} = \mathsf{true}$ holds, then it must hold that $\mathsf{restricted}_{\overline{u}}$ is true and $\tau_{r,\overline{u}}[r^{rest}_{\overline{u}}] = \tau_{s,u}[r^{rest}_{\overline{u}}]$ which cannot be the case because of lines 157 and 190 and the fact that $\mathsf{unchanged}_u$ is true. If $\mathsf{cheated}$ is true then either $(\sigma, v, \vec{\Delta}) = (\sigma^*[(r_u, \overline{u})], m^*[(r_u, \overline{u})], \vec{\Delta}^*[(r_u, \overline{u})])$ or $\vec{\Delta}'[(r_u, \overline{u})] \sqsubseteq \vec{\Delta}$ holds.

We start with the former case. The variables $\sigma^*[(r_u, \overline{u})]$, $m^*[(r_u, \overline{u})]$, and $\vec{\Delta}^*[(r_u, \overline{u})]$ would have been defined by the $\textsc{Sign}$ call on line 151 when $\overline{u}$ was sending a ciphertext and $s_{\overline{u}} = r_u + 1$. Variables $\sigma^*[(r_u, \overline{u})]$ and $m^*[(r_u, \overline{u})]$ thus uniquely define the ciphertext and associated data stored in $\mathbf{sctable}_u[s_{\overline{u}}]$ on line 155 (note that the ad component is uniquely defined because it is included as part of the ciphertext). So these variables equaling the σ and v returned by $\mathcal{A}_{\mathsf{DS}}$ when it aborts means that $\mathbf{sctable}_u[r_u+1] = (c, ad) \neq \mathbf{ctable}_u[r_u+1]$ where the inequality is from t_b holding in $\textsc{RecvSim}$. This implies that $\mathsf{restricted}_{\overline{u}}$ was true when $\mathbf{sctable}_u[r_u+1]$ and $\mathbf{ctable}_u[r_u+1]$ were set on lines 155 and 158 (so, in particular, $\mathbf{ctable}_u[r_u+1]$ was not set). We will show the following sequence of equalities holds $\tau_{r,\overline{u}}[r^{rest}_{\overline{u}}] = \vec{\Delta}^*[(r_u, \overline{u})][r^{rest}_{\overline{u}} - r^{sent}_{\overline{u}}] = \vec{\Delta}[r^{rest}_{\overline{u}} - r^{sent}_{\overline{u}}] = \tau_{s,u}[r^{rest}_{\overline{u}}]$, where $\vec{\Delta}^*[(r_u, \overline{u})]$ and $\vec{\Delta}$ are indexed starting from 1. The latter two equalities are immediate from our assumption that $\vec{\Delta}^*[(r_u, \overline{u})] = \vec{\Delta}$ and the way $\vec{\Delta}$ is defined on line 166. Because $\overline{u}$ was restricted during the $\textsc{SendSim}$ query that defined $\vec{\Delta}^*[(r_u, \overline{u})]$ it must

have held that $r_{\overline{u}}^{rest}$ was less than or equal to the value of $r_{\overline{u}}$. It must also have held that $r_{\overline{u}}^{rest}$ was strictly greater than $r_{\overline{u}}^{sent}$ because $\overline{u}$ was not restricted during their prior send operation (because u received this prior ciphertext and $\mathsf{unchanged}_u$ still holds). So during the receive operation that set $\mathsf{restricted}_{\overline{u}}$ and all other receive operations of $\overline{u}$ after they sent that prior ciphertext it could not have held that $sk_{\overline{u}} \neq \bot$ because that necessitates $\mathbf{forge}_u[r_u + 1] =$ "trivial", contradicting our assumption that $\mathcal{A}_{\mathsf{DS}}$ aborted. So line 180 will have resulted in $\vec{\Delta}_s[(r_u, \overline{u})][1, \ldots, r_{\overline{u}}^{rest} - r_{\overline{u}}^{sent}] = \tau_{r,\overline{u}}[r_{\overline{u}}^{sent} + 1, \ldots, r_{\overline{u}}^{rest}]$. Then the first equality follows because $\vec{\Delta}^*[(r_u, \overline{u})]$ is later set to equal $\vec{\Delta}_s[(r_u, \overline{u})]$ during the sending operation.

Consider the latter case that $\vec{\Delta}'[(r_u, \overline{u})] \sqsubseteq \vec{\Delta}$. Then $\vec{\Delta}'[(r_u, \overline{u})] \neq \bot$ so the adversary must have exposed $\overline{u}$ when $s_{\overline{u}}$ was equal to the current value of r_u. By assumption $\mathbf{forge}_u[r_u + 1] \neq$ "trivial" so this exposure must have been done when $\mathsf{restricted}_{\overline{u}}$ was true. The same arguments above apply to give that $\vec{\Delta}_s[(r_u, \overline{u})][1, \ldots, r_{\overline{u}}^{rest} - r_{\overline{u}}^{sent}] = \tau_{r,\overline{u}}[r_{\overline{u}}^{sent} + 1, \ldots, r_{\overline{u}}^{rest}]$. Then later during the exposure $\vec{\Delta}'[(r_u, \overline{u})]$ is set to equal $\vec{\Delta}_s[(r_u, \overline{u})]$. This results in the sequence of equalities $\tau_{r,\overline{u}}[r_{\overline{u}}^{rest}] = \vec{\Delta}'[(r_u, \overline{u})][r_{\overline{u}}^{rest} - r_{\overline{u}}^{sent}] = \vec{\Delta}[r_{\overline{u}}^{rest} - r_{\overline{u}}^{sent}] = \tau_{s,u}[r_{\overline{u}}^{rest}]$ from the assumption that $\vec{\Delta}'[(r_u, \overline{u})] \sqsubseteq \vec{\Delta}$ and how $\vec{\Delta}$ is defined.

Now we show $\tau_{r,\overline{u}}[r_{\overline{u}}^{rest}] = \tau_{s,u}[r_{\overline{u}}^{rest}]$. If $\vec{\Delta}'_{\overline{u}}[r_u] \sqsubseteq \vec{\Delta}$ holds, $\overline{u}$ must have been restricted before $\vec{\Delta}'_{\overline{u}}[r_u]$ was set (because t_a is true, so line 78 must not have been executed). Thus the upper bound index used to set $\vec{\Delta}'_{\overline{u}}[r_u]$ was at least $r_{\overline{u}}^{rest}$. The lower bound index was equal to $r_u^{ack} + 1$; from the argument about, $r_{\overline{u}}^{rest}$ is at least this large. So it must hold that $\tau_{r,\overline{u}}[r_{\overline{u}}^{rest}] = \tau_{s,u}[r_{\overline{u}}^{rest}]$.

Correct simulation by $\mathcal{A}_{\mathsf{DS}}$.

The correctness of $\mathcal{A}_{\mathsf{DS}}$ follows from the fact that none of $\mathcal{A}_{\mathsf{DS}}$'s oracles will abort early and return $\bot$. Recall again that $\mathcal{A}_{\mathsf{DS}}$ uses user identifiers of the form $\Lambda = (s_u, u)$.

Queries to NEWUSER are made in SENDSIM only after s_u has been incremented, so such queries will never be made for which $sk[\Lambda] \neq \bot$. For other oracles, note that $sk[(i, v)] = \bot$ will only hold if $i > s_v$ or if $\mathsf{nextop} = (v, \text{"send"})$ when the i-th ciphertext was sent by user v. The former case never occurs because oracle queries are only made with $i = s_u$ or $i = s_u - 1$.

The latter case corresponds to $\mathcal{D}$ having exposed the randomness underlying this sending call, so $\mathcal{A}_{\mathsf{DS}}$ simply samples the signature keys for itself. Then $sk_u \neq \bot$ so $\mathcal{A}_{\mathsf{DS}}$ will not make any further oracle queries with $\Lambda = (i, v)$.

The last possibility is $\mathcal{A}_{\mathsf{DS}}$ making a query $\textsc{Sign}(\Lambda, m)$ when $\vec{\Delta}^*[\Lambda] \neq \bot$. As with $\textsc{NewUser}$, queries to $\textsc{Sign}$ are made in $\textsc{SendSim}$ only after s_u has been incremented, so $\mathcal{A}_{\mathsf{DS}}$ will never make two signing queries with the same Λ implying that this possibility will not occur.

Claim 5, $\Pr[G_3] = \Pr[G_4]$**.**

To start the second stage of the proof consider games G_4 and G_5 shown in Fig. 2.16. We claim that the input-output behavior of G_4 is identical to G_3. We made three types of changes to G_3 to create G_4. First we generally cleaned things up by removing variables $\vec{\Delta}_u$, $\mathbf{rctable}_u$, $\mathbf{sctable}_u$, and r_u^{sent} which had no effect on the code and by simplifying the code to just abort when the if statements on lines 31, 52, or 68 evaluate to true, giving lines 222, 231, and 241. Next, we added the lines 232 through 235 for the transition to game G_5. Finally, we both replaced $\mathsf{unchanged}_u$ with (not $\mathsf{restricted}_u$) and removed the unnecessary parts of the if statements on lines 66 and 71 to obtain lines 240 and 242. These last changes require some justification.

First we claim that in G_3 when $\mathsf{unchanged}_u$ gets set false during a call to $\textsc{Recv}$, $\mathsf{restricted}_u$ will get set true during the same call. The converse is clear, so this establishes that they always have opposite truth values. The first time $\mathsf{unchanged}_u$ is set it must hold that $(c, ad) \neq \mathbf{ctable}_u[r_u]$ and that $\mathsf{unchanged}_u$ was true at the beginning of the execution of $\textsc{Recv}$. Then because the if statement on line 52 must have evaluated to false we have that $\mathbf{forge}_u[r_u] =$ "trivial" on line 66 so $\mathsf{restricted}_u$ will also be set.

For simplifying lines 66 and 71 note that if $(c, ad) \neq \mathbf{ctable}_u[r_u]$ then either $\mathsf{restricted}_u = \mathsf{true}$ already held at the beginning of this execution of $\textsc{Recv}$ or the if statement on line 52 implies that $\mathbf{forge}_u[r_u + 1] =$ "trivial" (before r_u has been incremented) so $\mathsf{restricted}_u$ will be set. This allows to simplify both if statements. Thus the fifth claim, $\Pr[G_3] = \Pr[G_4]$, holds.

```
Games G_4, G_5
206: b ←$ {0,1} ; r_I^rest ← r_R^rest ← ∞; s_I ← r_I ← s_R ← r_R ← 0; restricted_I ← restricted_R ← false
207: forge_I[0...∞] ← forge_R[0...∞] ← "nontrivial" ; X_I ← X_R ← 0
208: (sk_I, vk_R) ←$ DS.Kg ; (sk_R, vk_I) ←$ DS.Kg
209: (ek_I, dk_R[0]) ←$ PKE.Kg ; (ek_R, dk_I[0]) ←$ PKE.Kg
210: hk ←$ H.Kg ; τ_{r,I}[0] ← τ_{r,R}[0] ← τ_{s,I}[0] ← τ_{s,R}[0] ← ε ; r_I^ack ← r_R^ack ← 0
211: (z_I, z_R) ←$ (Ch.SendRS)^2 ; (η_I, η_R) ←$ (Ch.RecvRS)^2
212: b' ←$ D^{Send,Recv,Exp} ; Return (b' = b)

Send(u, m_0, m_1, ad)
213: If nextop ≠ (u, "send") and nextop ≠ ⊥ then return ⊥
214: If |m_0| ≠ |m_1| then return ⊥
215: If (r_u < X_u or restricted_u or ch_u[s_u + 1] = "forbidden") and m_0 ≠ m_1 then
216:     Return ⊥
217: (z_1, z_2, z_3, z_4) ← z_u ; s_u ← s_u + 1 ; (sk, vk) ← DS.Kg(z_1) ; (ek, dk_u[s_u]) ← PKE.Kg(z_2)
218: ℓ ← (s_u, r_u, ad, vk, ek, τ_{r,u}[r_u], τ_{s,u}[s_u − 1])
219: (ek', c') ← PKE.Enc(ek_u, ℓ, m_b, τ_{s,u}[r_u^ack + 1, ..., s_u − 1]; z_3)
220: v ← (c', ℓ) ; σ ← DS.Sign(sk_u, v; z_4) ; c ← (σ, v) ; τ_{s,u}[s_u] ← H.Ev(hk, c) ; sk_u ← sk
221: nextop ← ⊥ ; z_u ←$ Ch.SendRS
222: If not restricted_u and τ_{s,u}[s_u] = τ_{r,ū}[s_u] then abort(false)
223: If not restricted_u then ctable_ū[s_u] ← (c, ad)
224: If m_0 ≠ m_1 then ch_u[s_u] ← "done"
225: Return c

Recv(u, c, ad)
226: If nextop ≠ (u, "recv") and nextop ≠ ⊥ then return ⊥
227: (η_1, η_2) ← η_u ; nextop ← ⊥ ; η_u ←$ Ch.RecvRS
228: (σ, v) ← c ; (c', ℓ) ← v ; (s', r', ad', vk', ek', τ'_r, τ'_s) ← ℓ
229: (vk'', t) ← DS.Vrfy(vk_u, σ, v, τ_s[r_u^ack + 1, ..., r'])
230: If not t or s' ≠ r_u + 1 or τ'_r ≠ τ_{s,u}[r'] or τ'_s ≠ τ_{r,u}[r_u] or ad' ≠ ad then return ⊥
231: If not restricted_u and forge_u[r_u + 1] ≠ "trivial" and (c, ad) ≠ ctable_u[r_u + 1] then abort(false)
232: ((σ', v'), ad) ← ctable_u[r_u + 1]
233: If not restricted_u and v = v' and σ ≠ σ' then
234:     bad_uniq ← true
235:     abort(false)   / G_5
236: r_u ← r_u + 1 ; r_u^ack ← r' ; m ← PKE.Dec(dk_u[r_u^ack], ℓ, c')
237: τ_{s,u}[0, ..., r_u^ack − 1] ← ⊥ ; dk_u[0, ..., r_u^ack − 1] ← ⊥ ; τ_{r,u}[r_u] ← H.Ev(hk, c)
238: sk_u ← DS.UpdSk(sk_u, τ_{r,u}[r_u]; η_1) ; vk_u ← vk' ; ek_u ← ek'
239: For i ∈ [r_u^ack, s_u] do dk_u[i] ← PKE.UpdDk(dk_u[i], τ_{r,u}[r_u]; η_2^i)
240: If (c, ad) ≠ ctable_u[r_u] then restricted_u ← true ; r_u^rest ← min{r_u, r_u^rest}
241: If not restricted_ū and τ_{r,u}[r_u^rest] = τ_{s,ū}[r_u^rest] ≠ ⊥ then abort(false)
242: If restricted_u then return m
243: Return ⊥

Exp(u, rand)
244: If nextop ≠ ⊥ then return ⊥
245: If not restricted_u and ∃i ∈ (r_u, s_ū] s.t. ch_ū[i] = "done" then return ⊥
246: k⃗ ← (s_u, r_u, sk_u, vk_u, ek_u, dk_u, hk, τ_{r,u}[r_u], τ_{s,u})
247: If restricted_u then return (k⃗, z_u, η_u)
248: forge_ū[s_u + 1] ← "trivial" ; (z, η) ← (ε, ε) ; X_ū ← s_u + 1
249: If rand = "send" then
250:     nextop ← (u, "send") ; z ← z_u
251:     X_ū ← s_u + 2 ; forge_ū[s_u + 2] ← "trivial"
252:     ch_u[s_u + 1] ← "forbidden"
253: Else if rand = "recv" then
254:     nextop ← (u, "recv") ; η ← η_u
255: Return (k⃗, z, η)
```

Figure 2.16. Games G_4 and G_5 for security proof. Lines commented with the names of games are only included in those games. Highighted codes indicates changes from G_3.

Adversary $\mathcal{B}_{\mathsf{DS}}^{\text{NewUser}}$

256: $b \leftarrow\!\!\$ \{0,1\}$; $r_{\mathcal{I}}^{rest} \leftarrow r_{\mathcal{R}}^{rest} \leftarrow \infty$; $s_{\mathcal{I}} \leftarrow r_{\mathcal{I}} \leftarrow s_{\mathcal{R}} \leftarrow r_{\mathcal{R}} \leftarrow 0$
257: $\mathsf{restricted}_{\mathcal{I}} \leftarrow \mathsf{restricted}_{\mathcal{R}} \leftarrow \mathsf{false}$
258: $\mathbf{forge}_{\mathcal{I}}[0\ldots\infty] \leftarrow \mathbf{forge}_{\mathcal{R}}[0\ldots\infty] \leftarrow$ "nontrivial" ; $\mathcal{X}_{\mathcal{I}} \leftarrow \mathcal{X}_{\mathcal{R}} \leftarrow 0$
259: $z_{1,\mathcal{I}} \leftarrow \text{NewUser}((s_{\mathcal{I}},\mathcal{I}))$; $z_{1,\mathcal{R}} \leftarrow \text{NewUser}((s_{\mathcal{R}},\mathcal{R}))$
260: $(sk_{\mathcal{I}}, vk_{\mathcal{R}}) \leftarrow \mathsf{DS.Kg}(z_{1,\mathcal{I}})$; $(sk_{\mathcal{R}}, vk_{\mathcal{I}}) \leftarrow \mathsf{DS.Kg}(z_{1,\mathcal{R}})$
261: $(ek_{\mathcal{I}}, \mathbf{dk}_{\mathcal{R}}[0]) \leftarrow\!\!\$ \mathsf{PKE.Kg}$; $(ek_{\mathcal{R}}, \mathbf{dk}_{\mathcal{I}}[0]) \leftarrow\!\!\$ \mathsf{PKE.Kg}$
262: $hk \leftarrow\!\!\$ \mathsf{H.Kg}$; $\tau_{r,\mathcal{I}}[0] \leftarrow \tau_{r,\mathcal{R}}[0] \leftarrow \tau_{s,\mathcal{I}}[0] \leftarrow \tau_{s,\mathcal{R}}[0] \leftarrow \varepsilon$; $r_{\mathcal{I}}^{ack} \leftarrow r_{\mathcal{R}}^{ack} \leftarrow 0$
263: $z_{1,\mathcal{I}} \leftarrow \text{NewUser}((s_{\mathcal{I}},\mathcal{I}))$; $z_{1,\mathcal{R}} \leftarrow \text{NewUser}((s_{\mathcal{R}},\mathcal{R}))$
264: $(z_{2,\mathcal{I}}, z_{3,\mathcal{I}}, z_{4,\mathcal{I}}) \leftarrow\!\!\$ \mathsf{PKE.KgRS} \times \mathsf{PKE.EncRS} \times \mathsf{DS.SignRS}$
265: $(z_{2,\mathcal{R}}, z_{3,\mathcal{R}}, z_{4,\mathcal{R}}) \leftarrow\!\!\$ \mathsf{PKE.KgRS} \times \mathsf{PKE.EncRS} \times \mathsf{DS.SignRS}$
266: $z_{\mathcal{I}} \leftarrow (z_{1,\mathcal{I}}, z_{2,\mathcal{I}}, z_{3,\mathcal{I}}, z_{4,\mathcal{I}})$; $z_{\mathcal{R}} \leftarrow (z_{1,\mathcal{R}}, z_{2,\mathcal{R}}, z_{3,\mathcal{R}}, z_{4,\mathcal{R}})$
267: $(z_{\mathcal{I}}, z_{\mathcal{R}}) \leftarrow\!\!\$ (\mathsf{Ch.SendRS})^2$; $(\eta_{\mathcal{I}}, \eta_{\mathcal{R}}) \leftarrow\!\!\$ (\mathsf{Ch.RecvRS})^2$
268: $\mathcal{D}^{\text{SendSim},\text{RecvSim},\text{ExpSim}}$; Return $(\varepsilon,\varepsilon,\varepsilon,\varepsilon,(\varepsilon))$

$\text{SendSim}(u, m_0, m_1, ad)$

269: If $\mathsf{nextop} \neq (u,$"send"$)$ and $\mathsf{nextop} \neq \perp$ then return $\perp$
270: If $|m_0| \neq |m_1|$ then return $\perp$
271: If $(r_u < \mathcal{X}_u$ or $\mathsf{restricted}_u$ or $\mathbf{ch}_u[s_u+1] =$ "forbidden"$)$ and $m_0 \neq m_1$ then
272: Return $\perp$
273: $(z_1, z_2, z_3, z_4) \leftarrow z_u$; $s_u \leftarrow s_u + 1$
274: $(sk, vk) \leftarrow \mathsf{DS.Kg}(z_1)$; $(ek, \mathbf{dk}_u[s_u]) \leftarrow \mathsf{PKE.Kg}(z_2)$
275: $\ell \leftarrow (s_u, r_u, ad, vk, ek, \tau_{r,u}[r_u], \tau_{s,u}[s_u - 1])$
276: $(ek', c') \leftarrow \mathsf{PKE.Enc}(ek_u, \ell, m_b, \tau_{s,u}[r_u^{ack}+1, \ldots, s_u - 1]; z_3)$
277: $v \leftarrow (c', \ell)$; $\sigma \leftarrow \mathsf{DS.Sign}(sk_u, v; z_4)$
278: $c \leftarrow (\sigma, v)$; $\tau_{s,u}[s_u] \leftarrow \mathsf{H.Ev}(hk, c)$; $sk_u \leftarrow sk$; $\mathsf{nextop} \leftarrow \perp$;
279: $z_1 \leftarrow \text{NewUser}((s_u, u))$; $(z_2, z_3, z_4) \leftarrow\!\!\$ \mathsf{PKE.KgRS} \times \mathsf{PKE.EncRS} \times \mathsf{DS.SignRS}$
280: $z_u \leftarrow (z_1, z_2, z_3, z_4)$
281: If not $\mathsf{restricted}_u$ and $\tau_{s,u}[s_u] = \tau_{r,\overline{u}}[s_u]$ then $\mathbf{abort}(\varepsilon,\varepsilon,\varepsilon,\varepsilon,(\varepsilon))$
282: If not $\mathsf{restricted}_u$ then $\mathbf{ctable}_{\overline{u}}[s_u] \leftarrow (c, ad)$
283: If $m_0 \neq m_1$ then $\mathbf{ch}_u[s_u] \leftarrow$ "done"
284: Return c

$\text{RecvSim}(u, c, ad)$

285: If $\mathsf{nextop} \neq (u,$"recv"$)$ and $\mathsf{nextop} \neq \perp$ then return $\perp$
286: $(\eta_1, \eta_2) \leftarrow \eta_u$; $\mathsf{nextop} \leftarrow \perp$; $\eta_u \leftarrow\!\!\$ \mathsf{Ch.RecvRS}$
287: $(\sigma, v) \leftarrow c$; $(c', \ell) \leftarrow v$; $(s', r', ad', vk', ek', \tau_r', \tau_s') \leftarrow \ell$
288: $(vk'', t) \leftarrow \mathsf{DS.Vrfy}(vk_u, \sigma, v, \tau_s[r_u^{ack}+1, \ldots, r'])$
289: If not t or $s' \neq r_u + 1$ or $\tau_r' \neq \tau_{s,u}[r']$ or $\tau_s' \neq \tau_{r,u}[r_u]$ or $ad' \neq ad$ then return $\perp$
290: If not $\mathsf{restricted}_u$ and $\mathbf{forge}_u[r_u+1] \neq$ "trivial" and $(c, ad) \neq \mathbf{ctable}_u[r_u+1]$ then
291: Return $\mathbf{abort}(\varepsilon,\varepsilon,\varepsilon,\varepsilon,(\varepsilon))$
292: $((\sigma', v'), ad) \leftarrow \mathbf{ctable}_u[r_u+1]$
293: If not $\mathsf{restricted}_u$ and $v = v'$ and $\sigma \neq \sigma'$ then $\mathbf{abort}((r_u, \overline{u}), v, \sigma, \sigma', \tau_s[0, \ldots, r'])$
294: $r_u \leftarrow r_u + 1$; $r_u^{ack} \leftarrow r'$; $m \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}_u[r_u^{ack}], \ell, c')$
295: $\tau_{s,u}[0, \ldots, r_u^{ack} - 1] \leftarrow \perp$; $\mathbf{dk}_u[0, \ldots, r_u^{ack} - 1] \leftarrow \perp$; $\tau_{r,u}[r_u] \leftarrow \mathsf{H.Ev}(hk, c)$
296: $sk_u \leftarrow \mathsf{DS.UpdSk}(sk_u, \tau_{r,u}[r_u]; \eta_1)$; $vk_u \leftarrow vk'$; $ek_u \leftarrow ek'$
297: For $i \in [r_u^{ack}, s_u]$ do $\mathbf{dk}_u[i] \leftarrow \mathsf{PKE.UpdDk}(\mathbf{dk}_u[i], \tau_{r,u}[r_u]; \eta_2^i)$
298: If $(c, ad) \neq \mathbf{ctable}_u[r_u]$ then $\mathsf{restricted}_u \leftarrow \mathsf{true}$; $r_u^{rest} \leftarrow \min\{r_u, r_u^{rest}\}$
299: If not $\mathsf{restricted}_{\overline{u}}$ and $\tau_{r,u}[r_u^{rest}] = \tau_{s,\overline{u}}[r_u^{rest}] \neq \perp$ then $\mathbf{abort}(\varepsilon,\varepsilon,\varepsilon,\varepsilon,(\varepsilon))$
300: If $\mathsf{restricted}_u$ then return m
301: Return $\perp$

$\text{ExpSim}(u, \mathsf{rand})$

// Identical to Exp in G_4, G_5

Figure 2.17. Adversary $\mathcal{B}_{\mathsf{DS}}$ attacking DS. Highlighting indicates changes from G_4, G_5.

Claim 6, $\Pr[G_4] - \Pr[G_5] \leq \mathsf{Adv}^{\mathsf{uniq}}_{\mathsf{DS}}(\mathcal{B}_{\mathsf{DS}})$**.**

Games G_4 and G_5 are identical until bad_{uniq} so the fundamental lemma of game playing gives $\Pr[G_4] - \Pr[G_5] \leq \Pr[G_4 \text{ sets } \mathsf{bad}_{uniq}]$. Because bad_{uniq} only gets set if $\mathcal{D}$ has found a signature $\sigma' \neq \sigma$ which verifies correctly for v this can be transformed immediately to an attack on the UNIQ security of DS. This is shown by the adversary $\mathcal{B}_{\mathsf{DS}}$ in Fig. 2.17. It simulates the view of $\mathcal{D}$ by using NEWUSER to create the randomness used for digital signature key generation and computing everything else for itself. On line 293 it aborts and returns the signature collision found by $\mathcal{D}$ whenever bad_{uniq} would be set in G_4. It is clear that $\mathcal{B}_{\mathsf{DS}}$ correctly simulates the view of $\mathcal{D}$ and we will momentarily justify that it wins whenever it aborts, so $\Pr[G_4 \text{ sets } \mathsf{bad}_{uniq}] = \Pr[\mathrm{UNIQ}^{\mathcal{B}_{\mathsf{DS}}}_{\mathsf{DS}}] = \mathsf{Adv}^{\mathsf{uniq}}_{\mathcal{B}_{\mathsf{DS}}}(\mathsf{DS})$.

From line 293 we know that $\mathsf{restricted}_u = \mathsf{false}$ when $\mathcal{A}_{\mathsf{DS}}$ aborts, so vk_u is the same verification key used by UNIQ. Since $\mathbf{ctable}_u[r_u + 1]$ was not $\perp$, we know $\mathsf{restricted}_{\overline{u}}$ was false when this ciphertext was sent. This ensures that the sequence of strings used to update the signing key before this ciphertext was sent is the same sequence just used for verification.

Claim 7, $\Pr[G_5] \leq \Pr[\mathrm{INDEXP}^{\mathcal{A}_{\mathsf{PKE}}}_{\mathsf{PKE}}]$**.**

At last we can reduce directly to the security of the public key encryption scheme PKE. To do so we build an adversary against its security which simulates the view of $\mathcal{D}$ and forwards all of $\mathcal{D}$'s valid challenge queries to its own encryption oracle. Here we use the phrase "valid challenge query" to refer to calls that $\mathcal{D}$ makes to SEND for which $m_0 \neq m_1$ and G_5 would not return $\perp$. Then the adversary will output $\mathcal{D}$'s guess at the secret bit as its own (except when $\mathcal{D}$ causes an early abort, in which case the adversary simply returns 1). Such an adversary $\mathcal{A}_{\mathsf{PKE}}$ is shown in Fig. 2.18.

The adversary $\mathcal{A}_{\mathsf{PKE}}$ keeps track of the different keys it requests from INDEXP by labelling them with tuples of the form $\Lambda = (s, u)$, where u is the user who creates the key pair while sending a message and s was the value of s_u at that time. When $\mathcal{D}$ makes expose queries, $\mathcal{A}_{\mathsf{PKE}}$ makes calls to its EXP oracle as necessary to expose the correct state. Whenever a new

Adversary $\mathcal{A}_{\mathsf{PKE}}^{\textsc{NewUser},\textsc{UpdEk},\textsc{UpdDk},\textsc{Enc},\textsc{Dec},\textsc{Exp}}$

302: $r_{\mathcal{I}}^{rest} \leftarrow r_{\mathcal{R}}^{rest} \leftarrow \infty$; $s_{\mathcal{I}} \leftarrow r_{\mathcal{I}} \leftarrow s_{\mathcal{R}} \leftarrow r_{\mathcal{R}} \leftarrow 0$; $\mathsf{restricted}_{\mathcal{I}} \leftarrow \mathsf{restricted}_{\mathcal{R}} \leftarrow \mathsf{false}$
303: $\mathbf{forge}_{\mathcal{I}}[0\ldots\infty] \leftarrow \mathbf{forge}_{\mathcal{R}}[0\ldots\infty] \leftarrow$ "nontrivial" ; $\mathcal{X}_{\mathcal{I}} \leftarrow \mathcal{X}_{\mathcal{R}} \leftarrow 0$
304: $(sk_{\mathcal{I}}, vk_{\mathcal{R}}) \leftarrow\!\$\ \mathsf{DS.Kg}$; $(sk_{\mathcal{R}}, vk_{\mathcal{I}}) \leftarrow\!\$\ \mathsf{DS.Kg}$; $ek_{\mathcal{I}} \leftarrow \textsc{NewUser}((s_{\mathcal{R}}, \mathcal{R}))$; $ek_{\mathcal{R}} \leftarrow \textsc{NewUser}((s_{\mathcal{I}}, \mathcal{I}))$
305: $hk \leftarrow\!\$\ \mathsf{H.Kg}$; $\tau_{r,\mathcal{I}}[0] \leftarrow \tau_{r,\mathcal{R}}[0] \leftarrow \tau_{s,\mathcal{I}}[0] \leftarrow \tau_{s,\mathcal{R}}[0] \leftarrow \varepsilon$; $r_{\mathcal{I}}^{ack} \leftarrow r_{\mathcal{R}}^{ack} \leftarrow 0$
306: $(z_{\mathcal{I}}, z_{\mathcal{R}}) \leftarrow\!\$\ (\mathsf{Ch.SendRS})^2$; $(\eta_{\mathcal{I}}, \eta_{\mathcal{R}}) \leftarrow\!\$\ (\mathsf{Ch.RecvRS})^2$
307: $b' \leftarrow\!\$\ \mathcal{D}^{\textsc{SendSim},\textsc{RecvSim},\textsc{ExpSim}}$; Return b'

$\textsc{SendSim}(u, m_0, m_1, ad)$

308: If $\mathsf{nextop} \neq (u,$"send"$)$ and $\mathsf{nextop} \neq \bot$ then return $\bot$
309: If $|m_0| \neq |m_1|$ then return $\bot$
310: If ($r_u < \mathcal{X}_u$ or $\mathsf{restricted}_u$ or $\mathbf{ch}_u[s_u+1] =$ "forbidden") and $m_0 \neq m_1$ then
311: Return $\bot$
312: $(z_1, z_2, z_3, z_4) \leftarrow z_u$; $s_u \leftarrow s_u + 1$; $(sk, vk) \leftarrow \mathsf{DS.Kg}(z_1)$
313: If $\mathsf{nextop} \neq (u,$"send"$)$ and not $\mathsf{restricted}_u$ then $ek \leftarrow \textsc{NewUser}((s_u, u))$ else $(ek, \mathbf{dk}_u[s_u]) \leftarrow \mathsf{PKE.Kg}(z_2)$
314: $\ell \leftarrow (s_u, r_u, ad, vk, ek, \tau_{r,u}[r_u], \tau_{s,u}[s_u - 1])$; $\vec{\Delta}_e \leftarrow \tau_{s,u}[r_u^{ack}+1, \ldots, s_u - 1]$
315: If $m_0 \neq m_1$ then $c' \leftarrow \textsc{Enc}((r_u, \overline{u}), m_0, m_1, \ell)$ else $(ek', c') \leftarrow \mathsf{PKE.Enc}(ek_u, \ell, m_1, \vec{\Delta}_e; z_3)$
316: $v \leftarrow (c', \ell)$; $\sigma \leftarrow \mathsf{DS.Sign}(sk_u, v; z_4)$; $c \leftarrow (\sigma, v)$; $\tau_{s,u}[s_u] \leftarrow \mathsf{H.Ev}(hk, c)$; $sk_u \leftarrow sk$
317: If $r_u \geq \mathcal{X}_u$ and not $\mathsf{restricted}_u$ then $\textsc{UpdEk}((r_u, \overline{u}), \tau_{s,u}[s_u])$
318: $\mathsf{nextop} \leftarrow \bot$; $z_u \leftarrow\!\$\ \mathsf{Ch.SendRS}$
319: If not $\mathsf{restricted}_u$ and $\tau_{s,u}[s_u] = \tau_{r,\overline{u}}[s_u]$ then $\mathbf{abort}(1)$
320: If not $\mathsf{restricted}_u$ then $\mathbf{ctable}_{\overline{u}}[s_u] \leftarrow (c, ad)$
321: If $m_0 \neq m_1$ then $\mathbf{ch}_u[s_u] \leftarrow$ "done"
322: Return c

$\textsc{RecvSim}(u, c, ad)$

323: If $\mathsf{nextop} \neq (u,$"recv"$)$ and $\mathsf{nextop} \neq \bot$ then return $\bot$
324: $(\eta_1, \eta_2) \leftarrow \eta_u$; $\mathsf{nextop} \leftarrow \bot$; $\eta_u \leftarrow\!\$\ \mathsf{Ch.RecvRS}$
325: $(\sigma, v) \leftarrow c$; $(c', \ell) \leftarrow v$; $(s', r', ad', vk', ek', \tau_r', \tau_s') \leftarrow \ell$
326: $(vk'', t) \leftarrow \mathsf{DS.Vrfy}(vk_u, \sigma, v, \tau_s[r_u^{ack}+1, \ldots, r'])$
327: If not t or $s' \neq r_u + 1$ or $\tau_r' \neq \tau_{s,u}[r']$ or $\tau_s' \neq \tau_{r,u}[r_u]$ or $ad' \neq ad$ then return $\bot$
328: If not $\mathsf{restricted}_u$ and $\mathbf{forge}_u[r_u+1] \neq$ "trivial" and $(c, ad) \neq \mathbf{ctable}_u[r_u+1]$ then $\mathbf{abort}(1)$
329: $((\sigma', v'), ad) \leftarrow \mathbf{ctable}_u[r_u+1]$
330: If not $\mathsf{restricted}_u$ and $v = v'$ and $\sigma \neq \sigma'$ then $\mathbf{abort}(1)$
331: $r_u \leftarrow r_u + 1$; $r_u^{ack} \leftarrow r'$;
332: If $\mathbf{dk}_u[r_u^{ack}] = \bot$ then $m \leftarrow \textsc{Dec}((r_u^{ack}, u), c', \ell)$ else $m \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}_u[r_u^{ack}], \ell, c')$
333: $\tau_{s,u}[0, \ldots, r_u^{ack} - 1] \leftarrow \bot$; $\mathbf{dk}_u[0, \ldots, r_u^{ack} - 1] \leftarrow \bot$; $\tau_{r,u}[r_u] \leftarrow \mathsf{H.Ev}(hk, c)$
334: $sk_u \leftarrow \mathsf{DS.UpdSk}(sk_u, \tau_{r,u}[r_u]; \eta_1)$; $vk_u \leftarrow vk'$; $ek_u \leftarrow ek'$
335: If $r_u \geq \mathcal{X}_u$ and not $\mathsf{restricted}_u$ then for $i \in [r_u^{ack}+1, s_u]$ do $\textsc{UpdEk}((r_u, \overline{u}), \tau_{s,u}[i])$
336: For $i \in [r_u^{ack}, s_u]$ if $\mathbf{dk}_u[i] = \bot$ do $\textsc{UpdDk}((i, u), \tau_{r,u}[r_u])$ else $\mathbf{dk}_u[i] \leftarrow \mathsf{PKE.UpdDk}(\mathbf{dk}_u[i], \tau_{r,u}[r_u]; \eta_2^i)$
337: If $(c, ad) \neq \mathbf{ctable}_u[r_u]$ then $\mathsf{restricted}_u \leftarrow \mathsf{true}$; $r_u^{rest} \leftarrow \min\{r_u, r_u^{rest}\}$
338: If not $\mathsf{restricted}_{\overline{u}}$ and $\tau_{r,u}[r_u^{rest}] = \tau_{s,\overline{u}}[r_u^{rest}] \neq \bot$ then $\mathbf{abort}(1)$
339: If $\mathsf{restricted}_u$ then return m
340: Return $\bot$

$\textsc{ExpSim}(u, \mathsf{rand})$

341: If $\mathsf{nextop} \neq \bot$ then return $\bot$
342: If not $\mathsf{restricted}_u$ and $\exists i \in (r_u, s_{\overline{u}}]$ s.t. $\mathbf{ch}_{\overline{u}}[i] =$ "done" then
343: Return $\bot$
344: For $i \in [r_u^{ack}, s_u]$ if $\mathbf{dk}_u[i] = \bot$ then $\mathbf{dk}_u[i] \leftarrow \textsc{Exp}((i, u))$
345: $\vec{k} \leftarrow (s_u, r_u, sk_u, vk_u, ek_u, \mathbf{dk}_u, hk, \tau_{r,u}[r_u], \tau_{s,u})$
346: If $\mathsf{restricted}_u$ then return $(\vec{k}, z_u, \eta_u)$
347: $\mathbf{forge}_{\overline{u}}[s_u+1] \leftarrow$ "trivial" ; $(z, \eta) \leftarrow (\varepsilon, \varepsilon)$; $\mathcal{X}_{\overline{u}} \leftarrow s_u + 1$
348: If $\mathsf{rand} =$ "send" then
349: $\mathsf{nextop} \leftarrow (u,$"send"$)$; $z \leftarrow z_u$; $\mathcal{X}_{\overline{u}} \leftarrow s_u + 2$
350: $\mathbf{forge}_{\overline{u}}[s_u+2] \leftarrow$ "trivial" ; $\mathbf{ch}_u[s_u+1] \leftarrow$ "forbidden"
351: Else if $\mathsf{rand} =$ "recv" then
352: $\mathsf{nextop} \leftarrow (u,$"recv"$)$; $\eta \leftarrow \eta_u$
353: Return $(\vec{k}, z, \eta)$

Figure 2.18. Adversary $\mathcal{A}_{\mathsf{PKE}}$ attacking PKE. Highlighting indicates changes from $\mathrm{G}_4, \mathrm{G}_5$.

PKE key pair should be created, $\mathcal{A}_{\mathsf{PKE}}$ does so via a NEWUSER query unless the randomness for that key generation has been exposed, in which case it just generates the key pair by itself with said exposed randomness. Whenever $\mathcal{A}_{\mathsf{PKE}}$ needs the encryption of a message or decryption of a ciphertext, it will forward this on to the corresponding oracle in INDEXP except when it already has necessary information to perform this operation for itself.

Below we verify that adversary $\mathcal{A}_{\mathsf{PKE}}$ correctly simulates the view of $\mathcal{D}$ with the underlying bit of INDEXP playing the role of its secret bit. This gives $\Pr[\mathrm{G}_5] \leq \Pr[\mathrm{INDEXP}_{\mathsf{PKE}}^{\mathcal{A}_{\mathsf{PKE}}}]$, as desired.

Efficiency.

We have shown the stated bound on the advantage of $\mathcal{D}$. The bounds on the number of oracle queries made by the adversaries and their runtimes can be verified by examining their code.

Correct simulation by $\mathcal{A}_{\mathsf{PKE}}$.

We verify that none of the oracles called by $\mathcal{A}_{\mathsf{PKE}}$ will abort early and return $\perp$, except possibly DEC. We will argue that $\mathcal{A}_{\mathsf{PKE}}$ doesn't require the output of DEC to be non-$\perp$ in the cases when it returns $\perp$. Recall that $\mathcal{A}_{\mathsf{PKE}}$ uses user identifiers of the form $\Lambda = (s, u)$ for the key-pair created by u when sending their s-th ciphertext.

Queries to NEWUSER are made in SENDSIM only after s_u has been updated, so (since s_u is strictly increasing) such queries will never be made with $\mathbf{dk}[\Lambda] \neq \perp$. For other oracles, note that $\mathbf{dk}[(s, v)] = \perp$ will only hold if (i) $s > s_v$ or (ii) $(\mathsf{nextop} = (v, \text{“send”}) \text{ or } \mathsf{restricted}_v)$ was true when the s-th ciphertext was sent by v (this condition corresponding when $\mathcal{A}_{\mathsf{PKE}}$ simply samples the corresponding keys for itself). For purposes of analysis we can divide the oracle queries into two groups.

The first group is DEC, UPDDK, and EXP. For this group the oracle queries with $\Lambda = (s, v)$ are only made for $s \leq s_u$ so condition (i) will not hold. If condition (ii) held, then on line 313 $\mathbf{dk}_u[s]$ would have been set to a non-$\perp$ value so the checks before these three oracle

queries would prevent the queries from being made. Note here that line 333 can set entries of $\mathbf{dk}_u$ back to $\perp$, but only for entries with indices less than r_u^{ack} and these oracle queries are only made for indices at least as large as r_u^{ack} (which is a strictly increasing value).

The second group of oracles is ENC and UPDEK. Note that the latter can be called both by SENDSIM and by RECVSIM. All of these queries are done with $\Lambda = (r_u, \overline{u})$ and only if $r_u \geq \mathcal{X}_u$ and $\mathsf{restricted}_u = \mathsf{false}$. (In the case of ENC this is ensured by the fact that $m_0 \neq m_1$ and line 310 must have evaluated to false.) That $\mathsf{restricted}_u = \mathsf{false}$ ensures that $r_u \leq s_{\overline{u}}$, so condition (i) cannot hold. For the latter condition note that nextop can only be set to $(\overline{u}, \text{“send”})$ in EXP. But this would have set $\mathcal{X}_u$ such that $r_u \geq \mathcal{X}_u$ would be false. If $\mathsf{restricted}_{\overline{u}}$ held when the r_u-th ciphertext was sent by $\overline{u}$, then $\mathsf{restricted}_u$ would have been set when u received the corresponding ciphertext. So condition (ii) also cannot hold.

So none of the oracles will abort early due to checks involving **dk**. It remains to individually analyze the possibilities of ENC, DEC, or EXP aborting early for other reasons.

It will never be the case that $|m_0| \neq |m_1|$ in ENC because that would have resulted in SENDSIM aborting early. So consider the possibility that a call to ENC is made when $\vec{\Delta}'[(r_u, \overline{u})] \sqsubseteq \vec{\Delta}_e[(r_u, \overline{u})]$. Note that $\vec{\Delta}_e[(r_u, \overline{u})] = \tau_{s,u}[r_u^{ack}+1, \ldots, s_u - 1]$ and $\vec{\Delta}'[(r_u.\overline{u})] = \tau_{r,\overline{u}}[r_u^{ack}+1, \ldots, j]$ for some j. The value of $\vec{\Delta}'[(r_u, \overline{u})]$ must have been set by a call to EXP by EXPSIM when $r_{\overline{u}}$ was equal to j. If $\mathsf{restricted}_{\overline{u}}$ did not hold at the time, then $\mathcal{X}_u$ would have been set to a value greater than r_u so this ENC query would not be made. Also, $\mathsf{restricted}_u$ cannot hold when the ENC query is made. This implies that $\mathsf{restricted}_{\overline{u}}$ did not hold when the last ciphertext received by u was sent by $\overline{u}$. Thus, $r_u^{ack} < r_{\overline{u}}^{rest} \leq j$. Then $\vec{\Delta}'[(r_u, \overline{u})] \sqsubseteq \vec{\Delta}_e[(r_u, \overline{u})]$ would imply that $\tau_{s,u}[r_{\overline{u}}^{rest}] = \tau_{r,\overline{u}}[r_{\overline{u}}^{rest}]$ which is impossible. If $\tau_{r,\overline{u}}[r_{\overline{u}}^{rest}]$ was the first of these set to a non-$\perp$ value, then when $\tau_{s,u}[r_{\overline{u}}^{rest}]$ was set it would have caused $\mathcal{A}_{\mathsf{PKE}}$ to abort on line 319. If $\tau_{s,u}[r_{\overline{u}}^{rest}]$ was set first, then when $\tau_{r,\overline{u}}[r_{\overline{u}}^{rest}]$ was set it would have caused $\mathcal{A}_{\mathsf{PKE}}$ to abort on line 338.

Consider the possibility that a call to DEC is made when $(\vec{\Delta}_d[(r_u^{ack}, u)], c', \ell)$ is in the set $S[(r_u^{ack}, u)]$. Then there was a prior query of the form $\text{ENC}((r_u^{ack}, u), \cdot, \cdot, \ell)$ which returned c'. By logic we have used previously (using lines 315 and 310), $\mathsf{restricted}_{\overline{u}} = \mathsf{false}$ must have held at the

time. Then $\mathbf{ctable}_u[r_u]$ was set on line 320. If $\mathsf{restricted}_u = \mathsf{false}$ and $(c, ad) = \mathbf{ctable}_u[r_u]$, then $\mathsf{restricted}_u$ will still be false at the end of RECVSIM, so $\perp$ will be returned to $\mathcal{D}$ no matter what DEC outputs. If $\mathsf{restricted}_u = \mathsf{false}$ and $(c, ad) \neq \mathbf{ctable}_u[r_u]$, then from line 330 we can see that $v \neq v'$ which is a contradiction because both are equal to (c', ℓ). So suppose that $\mathsf{restricted}_u$ holds. Note that $\vec{\Delta}_d[(r_u^{ack}, u)] = \tau_{r,u}[j, \ldots, r_u - 1]$ where j is one more than the value r_u held when the the call NEWUSER$((r_u^{ack}, u))$ was made. Furthermore, $\vec{\Delta}_d[(r_u^{ack}, u)] = \tau_{s,\overline{u}}[j, \ldots, r_u - 1]$ because it is equal to the value $\vec{\Delta}_e[(r_u^{ack}, u)]$ held during the relevant ENC query. For DEC to have been queried it must hold that $\mathbf{dk}_u[r_u^{ack}] = \perp$, so $\mathsf{restricted}_u$ did not hold when NEWUSER$((r_u^{ack}, u))$ was queried. This means $r_u^{rest} \in [j, \ldots, r_u - 1]$ so we have $\tau_{r,u}[r_u^{rest}] = \tau_{s,\overline{u}}[r_u^{rest}]$. This is impossible by the same reasoning used in the prior paragraph.

Consider the possibility that a call to EXP is made when $\exists(\vec{\Delta}, c, \ell) \in S[(i, u)]$ such that $\vec{\Delta}_d[(i, u)] \sqsubseteq \vec{\Delta}$. Note that $\vec{\Delta}$ is equal to the value of $\vec{\Delta}_e[(i, u)]$ during some previous ENC query. By prior logic, $\mathsf{restricted}_{\overline{u}} = \mathsf{false}$ must have held at the time. Suppose $\mathsf{restricted}_u = \mathsf{false}$. By 342, it must hold that $\mathbf{ch}_{\overline{u}}[j] \neq$ "done" for all $j \in (r_u, s_{\overline{u}}]$ and note that $\mathbf{ch}_{\overline{u}}[s] =$ "done". Let s denote the value of $s_{\overline{u}}$ when the relevant ENC query was made. Note that $s \leq s_{\overline{u}}$ because $s_{\overline{u}}$ never decreases and $s > r_u$ because $|\vec{\Delta}_d[(i, u)]| \leq |\vec{\Delta}_e[(i, u)]|$. This is a contradiction. So suppose that $\mathsf{restricted}_u$ holds. Note that $\vec{\Delta}_d[(i, u)] = \tau_{r,u}[j, \ldots, r_u]$ where j is one more than the value r_u held when the the call NEWUSER$((r_u^{ack}, u))$ was made. Furthermore, $\vec{\Delta}_d[(r_u^{ack}, u)] = \tau_{s,\overline{u}}[j, \ldots, r_u]$ because it is equal to the prefix of $\vec{\Delta}_e[(i, u)]$ held during the relevant ENC query. For DEC to have been queried it must hold that $\mathbf{dk}_u[i] = \perp$, so $\mathsf{restricted}_u$ did not hold when NEWUSER$((i, u))$ was queried. This means $r_u^{rest} \in [j, \ldots, r_u]$ so we have $\tau_{r,u}[r_u^{rest}] = \tau_{s,\overline{u}}[r_u^{rest}]$. This is impossible by the same reasoning used two paragraphs prior. $\square$

2.6 Comparison to recent definitions

Three recents works we studied while deciding how to write our security definition were the works of CCDGS [29], BCJNS [17], and GM [43]. While they all ultimately were interested

in different settings (and CCDGS did not even model any form of encryption) they all share the commonality of modeling security in a setting where there are different "stages" of keys. All three works made distinct decisions in how to address challenges in different stages, so its worth discussing the repercussions of these different decisions.

In BCJNS and GM a stage corresponds to a counter i representing how many times the relevant key has been updated. In CCDGS a stage corresponds to a tuple (u, j, i) corresponding to user u in step i of its j-th protocol execution. In our work a stage could correspond to a tuple (u, O, s, r) corresponding to a query to user $u \in \{\mathcal{I}, \mathcal{R}\}$ with oracle $\mathrm{O} \in \{\textsc{Send}, \textsc{Recv}\}$ when $s_u = s$ and $r_u = r$.

CCDGS chose to only allow the adversary to make a challenge query in a single stage for which there is a single corresponding bit that it must guess. BCJNS also has only a single challenge bit, but allows the adversary to make challenge queries in arbitrarily many stages all of which share that bit. GM also allows challenge queries in arbitrarily many stages, but samples a separate challenge bit for each stage. At the end of the game the adversary outputs both a bit and the index of the stage for which it is trying to guess the bit. BCJNS thus needed to keep track throughout of whether an adversary has "cheated" by doing something it is not allowed to which would tell it was the secret bit was. CCDGS and GM only need to perform this check for the particular stage for which the adversary attempts to guess the bit. This check is still somewhat global because actions in other stages may affect whether the adversary has cheated for the challenge stage. We will refer to these different styles of definitional choices as CCDGS, BCJNS, or GM security and discuss them broadly without regard to how stages are defined or what the underlying game is.

Qualitatively these three definitional choices result in equivalent definitions. BCJNS security and GM security can both easily be shown to imply CCDGS security. In the other direction, a hybrid argument can be used to show that CCDGS security implies BCJNS security and an index guessing proof can be used to show that it implies GM security. However, both of these proof techniques introduce a factor q loss of security (where q is the maximum number of

stages an adversary ever interacts with) so CCDGS security appears to be quantitively weaker than the other two. BCJNS and GM security appear to be quantitatively incomparable; the only way we are aware of to show that one implies the other is to use CCDGS security as an intermediate step which introduces a factor q loss of security in *both* directions.

BCJNS security challenges the adversary to learn one bit corresponding exactly to which of two possible “worlds” it exists in. GM security instead challenges the adversary to learn one bit about the exponential number of “worlds” it may exist in (though, to be clear, it can choose to cheat to learn all of the bits of information about which “world” it is in other than the one bit it attempts to guess). BCJNS show that their style of definition allows tight security reductions from multi-user security assumptions about the underlying primitives. GM did not aim to give tight security notions, but it is likely that tight reductions could be given from a variant multi-user style assumption in which the game samples an independent challenge bit for every user and the adversary wins if it can guess the bit corresponding to a single user. We are not aware of any works that use such a multi-user definition directly and do not know if techniques used to tightly prove security for standard multi-user definition would extend to such a definition.

This multi-user style definition is similar to multi-instance style definitions as introduced by [13] which sample an independent bit for each user and require the adversary guess the xor of *all* the challenge bits. They give away the underlying challenge bit when the secrets of a user are exposed, so considering adversaries who expose all users except one gives a notion essentially equivalent to the multi-user notion sketched above.

When considering the above, we ultimately decided to follow a BCJNS style of security definition and provide tight reductions from the standard multi-user security of underlying primitives.

2.7 Security implications of correctness notions

The correctness of a channel will have security implications outside of the scope of what we capture in our coming formalism. We suggest that CORR correctness may be a more appropriate notion of correctness in the secure messaging setting, but also note some plausible scenarios that could be applicable in the secure messaging setting for which CORR$\perp$ is more appropriate.

Recall that the correctness required by CORR$\perp$ is identical to that of Marson and Poettering [49][4] and is the standard notion for channels. It follows a common convention that a user will permanently refuse to send or receive future ciphertexts once they have received an invalid ciphertext. In practice this then requires that the connection be re-established for communication to continue. In secure messaging, securely re-establishing a connection is typically quite costly because it requires out-of-band human interaction. The CORR notion of correctness instead uses a form of robustness (analogous to that of [17]) to avoid that requirement. Thus this correctness prevents a denial-of-service attack by which an adversary without any secret knowledge can send a single incorrect ciphertext to either party to kill the communication channel.

Against a stronger adversary the correctness required by CORR$\perp$ prevents a worse attack. Suppose an attacker can compromise the state of user u. Then it can impersonate u and send messages to $\overline{u}$. Under CORR$\perp$ correctness the next time u attempts to send a message to $\overline{u}$ if the attacker cannot block this communication then this will permanently kill the channel and the users will know something has gone wrong. Under CORR correctness, this extra communication would simply be silently dropped. The users would need to communicate out-of-band to realize something had gone wrong (perhaps when u notices that $\overline{u}$ has stopped replying to any of its messages). For this reason an application using CORR correct should likely warn the user when

[4]This holds when requiring perfect security, which both works do. The two notions would be distinct if we considered computational correctness.

it silently drops a message.

If the attacker even more powerful and able to block the outgoing communication sent by u from reaching $\overline{u}$ then the distinction between the two notions goes away. The users have to again fall back on out-of-band communication of the compromised status when u realizes that $\overline{u}$ is not receiving its messages.

The specification of the Double Ratchet Algorithm states *"If an exception is raised (e.g. message authentication failure) then the message is discarded and changes to the state object are discarded."* [36, Section 3.5]. This indicates that it would satisfy CORR. However, the Double Ratchet Algorithm was clearly designed to achieve a stricter notion of correctness because it intentionally accepts ciphertexts *in any order*. This goes against the standard security requirement that ciphertexts should only be accepted in the order they were sent, so we intentionally do not provide a correctness notion that captures this.

2.8 Construction of key-updatable digital signatures

In this section we formally specify the construction of a key-updatable digital signature scheme that we sketched in Section 2.2.1. For this purpose we use a key-evolving digital signature scheme.

Key-evolving digital signature schemes.

A key-evolving digital signature scheme is a digital signature scheme with an additional algorithm $\mathsf{DS.Up}$ and with a modified verification algorithm $\mathsf{DS.Vrfy}$. Update algorithm $\mathsf{DS.Up}$ takes a signing key sk to return an updated signing key, denoted by $sk \leftarrow\$ \mathsf{DS.Up}(sk)$. Modified verification algorithm $\mathsf{DS.Vrfy}$ takes verification key vk, signature σ, message m, and time period $n \in \mathbb{N}$ to return a decision $t \in \{\mathsf{true}, \mathsf{false}\}$ regarding whether σ is a valid signature of m under vk for the n-th secret key, denoted by $t \leftarrow \mathsf{DS.Vrfy}(vk, \sigma, m, n)$.

Correctness is defined by game DSCORR2 in Fig. 2.19. For adversary $\mathcal{C}$ we define $\mathsf{Adv}^{\mathsf{dscorr2}}_{\mathsf{DS}}(\mathcal{C}) = \Pr[\text{DSCORR2}^{\mathcal{C}}_{\mathsf{DS}}]$ and require that $\mathsf{Adv}^{\mathsf{dscorr2}}_{\mathsf{DS}}(\mathcal{C}) = 0$ for all (even unbounded)

Game $\mathrm{DSCORR2}_{\mathsf{DS}}^{\mathcal{C}}$	Game $\mathrm{FSUF}_{\mathsf{DS}}^{\mathcal{A}_{\mathrm{KE}}}$
$\nu \leftarrow\$ \mathsf{DS.KgRS}$; $n \leftarrow 1$	$S \leftarrow \emptyset$; $i \leftarrow 1$
$(sk, vk) \leftarrow \mathsf{DS.Kg}(\nu)$	$(sk, vk) \leftarrow\$ \mathsf{DS.Kg}$
$\mathcal{C}^{\mathrm{UP,SIGN}}(\nu)$	$(\sigma, m, n) \leftarrow\$ \mathcal{A}_{\mathrm{KE}}^{\mathrm{UP,SIGN,EXP}}(vk)$
Return bad	$t_1 \leftarrow ((\sigma, m, n) \in S)$
$\underline{\mathrm{UP}()}$	$t_2 \leftarrow$ exposed and $(n^* \leq n)$
$n \leftarrow n+1$	cheated $\leftarrow (t_1$ or $t_2)$
$sk \leftarrow\$ \mathsf{DS.Up}(sk)$	win $\leftarrow \mathsf{DS.Vrfy}(vk, \sigma, m, n)$
Return sk	Return win and not cheated
$\underline{\mathrm{SIGN}(m)}$ // $m \in \{0,1\}^*$	$\underline{\mathrm{UP}()}$
$\sigma \leftarrow\$ \mathsf{DS.Sign}(sk, m)$	$i \leftarrow i+1$
$t \leftarrow \mathsf{DS.Vrfy}(vk, \sigma, m, n)$	$sk \leftarrow\$ \mathsf{DS.Up}(sk)$
If not t then bad $\leftarrow$ true	Return $\perp$
	$\underline{\mathrm{SIGN}(m)}$ // $m \in \{0,1\}^*$
Game $\mathrm{FSUNIQ}_{\mathsf{DS}}^{\mathcal{B}_{\mathrm{KE}}}$	$\sigma \leftarrow\$ \mathsf{DS.Sign}(sk, m)$
$z \leftarrow\$ \mathsf{DS.KgRS}$	$S \leftarrow S \cup \{(\sigma, m, i)\}$
$(sk, vk) \leftarrow \mathsf{DS.Kg}(z)$	Return σ
$(m, \sigma_1, \sigma_2, n) \leftarrow\$ \mathcal{B}_{\mathrm{KE}}(z)$	$\underline{\mathrm{EXP}()}$
$t_1 \leftarrow \mathsf{DS.Vrfy}(vk, \sigma_1, m, n)$	$n^* \leftarrow i$; exposed $\leftarrow$ true
$t_2 \leftarrow \mathsf{DS.Vrfy}(vk, \sigma_2, m, n)$	Return sk
Return t_1 and t_2 and $\sigma_1 \neq \sigma_2$	

Figure 2.19. Games defining correctness, uniqueness, and forward security of key-evolving digital signature scheme DS.

adversaries.

Forward-secure signatures.

Forward security of a key-evolving digital signature scheme asks that, even if the key is exposed in some time period n^*, it should be computationally hard to forge a valid signature for any prior time period $n < n^*$. Our definition follows that of [10]. Formally, consider game FSUF shown in Fig. 2.19, associated to a key-evolving digital signature scheme DS and an adversary $\mathcal{A}_{\mathrm{KE}}$. The game generates a digital signature key pair for the initial time period $i = 1$ and runs the adversary with the verification key as input. The goal of the adversary is to forge a signature for an arbitrary time period n. The adversary is provided with access to oracles UP, SIGN and EXP. Oracle UP is used to advance into the next time period, incrementing its index i and updating

Algorithm $\mathsf{DS_{KU}.Kg}$	Algorithm $\mathsf{DS_{KU}.Sign}(sk,m)$
$(sk_{\mathrm{KE}}, vk_{\mathrm{KE}}) \leftarrow\$ \mathsf{DS_{KE}.Kg}$ $\Sigma[1,\ldots,\infty] \leftarrow \varepsilon$; $\vec{\Delta}[1,\ldots,\infty] \leftarrow \varepsilon$ $sk \leftarrow (sk_{\mathrm{KE}}, 1, \Sigma)$; $vk \leftarrow (vk_{\mathrm{KE}}, 1, \vec{\Delta})$ Return (sk, vk)	$(sk_{\mathrm{KE}}, i, \Sigma) \leftarrow sk$ $\sigma_m \leftarrow\$ \mathsf{DS_{KE}.Sign}(sk_{\mathrm{KE}}, 1 \,\|\, m)$ $\sigma \leftarrow (\sigma_m, i, \Sigma)$ Return σ
Algorithm $\mathsf{DS_{KU}.Vrfy}(vk,\sigma,m)$	Algorithm $\mathsf{DS_{KU}.UpdSk}(sk,\Delta)$
$(vk_{\mathrm{KE}}, i_{\max}, \vec{\Delta}) \leftarrow vk$ $(\sigma_m, i, \Sigma) \leftarrow \sigma$ If $i \neq i_{\max}$ then return false $t \leftarrow \mathsf{DS_{KE}.Vrfy}(vk_{\mathrm{KE}}, \sigma_m, 1 \,\|\, m, i)$ For $j = 1,\ldots,(i-1)$ do $t \leftarrow t$ and $\mathsf{DS_{KE}.Vrfy}(vk_{\mathrm{KE}}, \Sigma[j], 0 \,\|\, \vec{\Delta}[j], j)$ Return t	$(sk_{\mathrm{KE}}, i, \Sigma) \leftarrow sk$ $\Sigma[i] \leftarrow\$ \mathsf{DS_{KE}.Sign}(sk_{\mathrm{KE}}, 0 \,\|\, \Delta)$ $sk_{\mathrm{KE}} \leftarrow\$ \mathsf{DS_{KE}.Up}(sk_{\mathrm{KE}})$ $sk \leftarrow (sk_{\mathrm{KE}}, i+1, \Sigma)$ Return sk Algorithm $\mathsf{DS_{KU}.UpdVk}(vk,\Delta)$ $(vk_{\mathrm{KE}}, i, \vec{\Delta}) \leftarrow vk$; $\vec{\Delta}[i] \leftarrow \Delta$ $vk \leftarrow (vk_{\mathrm{KE}}, i+1, \vec{\Delta})$ Return vk

Figure 2.20. Key-updatable digital signature scheme $\mathsf{DS_{KU}} = \mathsf{DS\text{-}CONS}[\mathsf{DS_{KE}}]$.

---

the secret key of the scheme accordingly. Oracle SIGN uses the current signing key to return a signature σ for an arbitrary message m (for current time period i); note that adversary is not allowed to win the game by returning (σ, m, i) as its forgery. Finally, oracle EXP exposes the current signing key. Without loss of generality, the adversary can only call this oracle once. If the exposure happened in time period n^*, then the adversary is only considered to win the game if it returns a forgery for some time period $n < n^*$, meaning it cannot use the exposed key to trivially win the game. The advantage of $\mathcal{A}_{\mathrm{KE}}$ in breaking the FSUF security of DS is $\mathsf{Adv}^{\mathsf{fsuf}}_{\mathsf{DS}}(\mathcal{A}_{\mathrm{KE}}) = \Pr[\mathrm{FSUF}^{\mathcal{A}_{\mathrm{KE}}}_{\mathsf{DS}}]$.

Signature uniqueness.

Uniqueness of a key-evolving digital signature scheme requires that an adversary cannot find two distinct signatures that verify for the same message and the same round n. Consider the game FSUNIQ shown in Fig. 2.19, associated to a key-evolving signature scheme DS and an adversary $\mathcal{B}_{\mathrm{KE}}$. The advantage of $\mathcal{B}_{\mathrm{KE}}$ in breaking the FSUNIQ security of DS is $\mathsf{Adv}^{\mathsf{fsuniq}}_{\mathsf{DS}}(\mathcal{B}_{\mathrm{KE}}) = \Pr[\mathrm{FSUNIQ}^{\mathcal{B}_{\mathrm{KE}}}_{\mathsf{DS}}]$.

Key-updatable digital signature scheme $\mathsf{DS\text{-}CONS}[\mathsf{DS}_{\mathrm{KE}}]$**.**

Let $\mathsf{DS}_{\mathrm{KE}}$ be a key-evolving digital signature scheme. We build a key-updatable digital signature scheme $\mathsf{DS}_{\mathrm{KU}} = \mathsf{DS\text{-}CONS}[\mathsf{DS}_{\mathrm{KE}}]$ as defined in Fig. 2.20, where $\mathsf{DS}_{\mathrm{KU}}.\mathsf{KgRS} = \mathsf{DS}_{\mathrm{KE}}.\mathsf{KgRS}$ and $\mathsf{DS}_{\mathrm{KU}}.\mathsf{SignRS} = \mathsf{DS}_{\mathrm{KE}}.\mathsf{SignRS}$.

The key-updatable digital signature scheme $\mathsf{DS}_{\mathrm{KU}}$ utilizes a key-evolving signature scheme $\mathsf{DS}_{\mathrm{KE}}$ in the following way. The signing key sk of $\mathsf{DS}_{\mathrm{KU}}$ consists of the corresponding signing key sk_{KE} of $\mathsf{DS}_{\mathrm{KE}}$, along with an empty list Σ. In order to update the signing key sk of $\mathsf{DS}_{\mathrm{KU}}$ with a sequence of labels $\Delta_1, \Delta_2, \ldots, \Delta_q$, the scheme consecutively signs $\sigma_j \leftarrow\$ \mathsf{DS}_{\mathrm{KE}}.\mathsf{Sign}(sk_{\mathrm{KE}}, 0 \,\|\, \Delta_j)$ with the underlying signing key sk_{KE} of $\mathsf{DS}_{\mathrm{KE}}$ and updates sk_{KE} using algorithm $\mathsf{DS}_{\mathrm{KE}}.\mathsf{Up}$, for each $j = 1, \ldots, q$. The resulting signing key sk of scheme $\mathsf{DS}_{\mathrm{KU}}$ is defined to contain the derived signing key sk_{KE} of scheme $\mathsf{DS}_{\mathrm{KE}}$ along with the produced signatures $\Sigma = (\sigma_1, \sigma_2, \ldots)$ as per above.

The verification key vk of $\mathsf{DS}_{\mathrm{KU}}$ initially contains the corresponding verification key vk_{KE} of scheme $\mathsf{DS}_{\mathrm{KE}}$, along with an empty list $\vec{\Delta}$. To update vk with Δ, the key update Δ is appended to $\vec{\Delta}$.

The signature of a message m for key $sk = (sk_{\mathrm{KE}}, \Sigma)$ is defined as (σ_m, Σ) for $\sigma_m \leftarrow\$ \mathsf{DS}_{\mathrm{KE}}.\mathsf{Sign}(sk_{\mathrm{KE}}, 1 \,\|\, m)$. Let $\Sigma = (\sigma_1, \sigma_2, \ldots \sigma_{\kappa-1})$ and let $\vec{\Delta} = (\Delta_1, \Delta_2, \ldots, \Delta_{\kappa-1})$ for some κ. To verify the $\mathsf{DS}_{\mathrm{KU}}$ signature (σ_m, Σ) against message m for verification key $vk = (vk_{\mathrm{KE}}, \vec{\Delta})$, one has to check that σ_i is a valid $\mathsf{DS}_{\mathrm{KE}}$ signature for message Δ_i for time period i, for each $i = 1, \ldots, \kappa - 1$, and that σ_m is a valid signature for m in time period κ.

Security of $\mathsf{DS\text{-}CONS}[\mathsf{DS}_{\mathrm{KE}}]$**.**

Consider $\mathsf{DS}_{\mathrm{KU}} = \mathsf{DS\text{-}CONS}[\mathsf{DS}_{\mathrm{KE}}]$ for any key-evolving digital signature scheme $\mathsf{DS}_{\mathrm{KE}}$. We claim that for any adversary attacking UNIQ (signature uniqueness) or UFEXP (unforgeability under exposures) of $\mathsf{DS}_{\mathrm{KU}}$, there is an adversary that breaks FSUNIQ (signature uniqueness) or FSUF (forward security) of $\mathsf{DS}_{\mathrm{KE}}$, respectively, using roughly the same number of oracle queries and with roughly the same time efficiency.

Theorem 10. *Let* $\mathsf{DS}_{\mathrm{KU}} = \mathsf{DS\text{-}CONS}[\mathsf{DS}_{\mathrm{KE}}]$ *where* $\mathsf{DS}_{\mathrm{KE}}$ *be a key-evolving digital signature scheme. Let* $\mathcal{B}_{\mathsf{DS}}$ *be an adversary making at most* 1 *query to its* NEWUSER *oracle. Then we can build adversary* $\mathcal{B}_{\mathrm{KE}}$ *such that*

$$\mathsf{Adv}^{\mathsf{uniq}}_{\mathsf{DS}_{\mathrm{KU}}}(\mathcal{B}_{\mathsf{DS}}) \leq \mathsf{Adv}^{\mathsf{fsuniq}}_{\mathsf{DS}_{\mathrm{KE}}}(\mathcal{B}_{\mathrm{KE}}). \tag{2.1}$$

The runtime of $\mathcal{B}_{\mathrm{KE}}$ *is about that of* $\mathcal{B}_{\mathsf{DS}}$.

Theorem 11. *Let* $\mathsf{DS}_{\mathrm{KE}}$ *be a key-evolving digital signature scheme. Additionally, let* $\mathsf{DS}_{\mathrm{KU}} = \mathsf{DS\text{-}CONS}[\mathsf{DS}_{\mathrm{KE}}]$. *Let* $\mathcal{A}_{\mathsf{DS}}$ *be an adversary making at most* 1 *query to its* NEWUSER *oracle,* q_{UPD} *queries to its* UPD *oracle,* 1 *query to its* SIGN *oracle, and* 1 *queries to its* EXP *oracle. Then we can build adversary* $\mathcal{A}_{\mathrm{KE}}$ *such that*

$$\mathsf{Adv}^{\mathsf{ufexp}}_{\mathsf{DS}_{\mathrm{KU}}}(\mathcal{A}_{\mathsf{DS}}) \leq \mathsf{Adv}^{\mathsf{fsuf}}_{\mathsf{DS}_{\mathrm{KE}}}(\mathcal{A}_{\mathrm{KE}}). \tag{2.2}$$

Adversary $\mathcal{A}_{\mathrm{KE}}$ *makes at most* q_{UPD} *queries to its* UP *oracle,* $q_{\mathrm{UPD}}+1$ *queries to its* SIGN *oracle, and* 1 *query to its* EXP *oracle. The runtime of* $\mathcal{A}_{\mathrm{KE}}$ *is about that of* $\mathcal{A}_{\mathsf{DS}}$.

For simplicity, our theorems are given for single-user security (only one query to NEWUSER). A standard argument implies that our theorems extend to the multi-user security in a straightforward way with the same advantage bounds (using multi-user security of the underlying key-evolving digital signature scheme). Furthermore, since we consider only a single user, we can assume without loss of generality that adversary $\mathcal{A}_{\mathsf{DS}}$ in Theorem 11 makes only one query to oracle SIGN.

The formal proofs of the above theorems are straightforward, so we omit them. For Theorem 10 note that a signature from $\mathsf{DS}_{\mathrm{KU}}$ essentially just consists of a sequence of signatures from $\mathsf{DS}_{\mathrm{KE}}$, specifically a signature $\Sigma[i]$ for each Δ_i and a final σ_m for the actual message. If the two $\mathsf{DS}_{\mathrm{KU}}$ signatures are distinct either they differ in σ_m or in some $\Sigma[i]$. Wherever they differ we immediately have two different $\mathsf{DS}_{\mathrm{KE}}$ signatures that verify for the same string. The proof is straightforward, so we omit it.

To prove Theorem 11, note that adversary $\mathcal{A}_{\mathrm{KE}}$ (playing game FSUF) can perfectly

simulate game UFEXP for adversary $\mathcal{A}_{\mathsf{DS}}$, answering the UPD, SIGN, EXP oracle queries of the latter using the oracles UP, SIGN, EXP of the former, respectively. As with our description above, whenever $\mathcal{A}_{\mathsf{DS}}$ successfully forges on $\mathsf{DS}_{\mathrm{KU}}$ we immediately get a forgery on $\mathsf{DS}_{\mathrm{KE}}$ by considering these sub-signatures.

2.9 Construction of key-updatable public-key encryption

In this section we more formally specify the construction of a key-updatable public-key encryption scheme that we sketched in Section 2.2.2.

Hierarchical identity based encryption.

In a hierarchical identity based encryption scheme, or HIBE [40], there are decryption keys (referred to as identity keys) associated with identities which are tuples of strings. Every user of the scheme can choose to encrypt to any identity knowing only that identity and some public parameters. Then anybody with an identity key for that identity can decrypt the ciphertext. Furthermore, the hierarchical part of HIBE means that any user with an identity key K for identity $\vec{I}$ can delegate a sub-key for any identity $\vec{I}'$ such that $\vec{I}$ is a prefix of $\vec{I}'$ (i.e. $\vec{I} \sqsubseteq \vec{I}'$).

More formally, HIBE scheme HIBE specifies algorithms $\mathsf{HIBE.Set}$, $\mathsf{HIBE.Del}$, $\mathsf{HIBE.Enc}$, and $\mathsf{HIBE.Dec}$. Setup algorithm $\mathsf{HIBE.Set}$ returns the public parameters pp and identity key K for the identity $\vec{I} = ()$. We write $(pp, K) \leftarrow\$ \mathsf{HIBE.Set}$. Delegation algorithm $\mathsf{HIBE.Del}$ takes as input identity key K for the identity $\vec{I}$ and string $I \in \{0,1\}^*$ to return identity key K' for identity $\vec{I} \| I$. We write $K' \leftarrow\$ \mathsf{HIBE.Del}(K, I)$. Encryption algorithm $\mathsf{HIBE.Enc}$ takes as input public parameters pp, identity $\vec{I}$, label ℓ, and message m to produce ciphertext c. We write $c \leftarrow\$ \mathsf{HIBE.Enc}(pp, \vec{I}, \ell, m)$. Decryption algorithm $\mathsf{HIBE.Dec}$ takes as input identity key K, label ℓ, and ciphertext c to produce message $m \in \{0,1\}^*$. We write $m \leftarrow \mathsf{HIBE.Dec}(K, \ell, c)$. We let $\mathsf{HIBE.EncRS}$ denote the denote the set from which $\mathsf{HIBE.Enc}$ draws its random coins.

We denote the min-entropy of algorithms $\mathsf{HIBE.Set}$ and $\mathsf{HIBE.Enc}$ by $\mathrm{H}_\infty(\mathsf{HIBE.Set})$ and

$\underline{\text{Game HIBECORR}^{\mathcal{C}}_{\mathsf{HIBE}}}$
$(pp, K) \leftarrow\!\!\$\ \mathsf{HIBE.Set}\,;\ \vec{I} \leftarrow ()$
$\mathcal{C}^{\textsc{Delegate},\textsc{Enc}}(pp, K)$
Return bad

$\underline{\textsc{Delegate}(I)}$ $\quad$ **/** $I \in \{0,1\}^*$
$K \leftarrow\!\!\$\ \mathsf{HIBE.Del}(K, I)\,;\ \vec{I} \leftarrow \vec{I} \,\|\, I$
Return K

$\underline{\textsc{Enc}(\ell, m)}$ $\quad$ **/** $\ell, m \in \{0,1\}^*$
$c \leftarrow\!\!\$\ \mathsf{HIBE.Enc}(pp, \vec{I}, \ell, m)$
$m' \leftarrow \mathsf{HIBE.Dec}(K, \ell, c)$
If $m' \neq m$ then bad $\leftarrow$ true

Figure 2.21. Games defining correctness of hierarchical public-key encryption scheme HIBE.

$\mathrm{H}_\infty(\mathsf{HIBE.Enc})$, respectively, defined as follows:

$$2^{-\mathrm{H}_\infty(\mathsf{HIBE.Set})} = \max_{pp} \Pr\left[pp^* = pp : (pp^*, K^*) \leftarrow\!\!\$\ \mathsf{HIBE.Set}\right],$$

$$2^{-\mathrm{H}_\infty(\mathsf{HIBE.Enc})} = \max_{pp, \vec{I}, \ell, m, c} \Pr\left[c^* = c : c^* \leftarrow\!\!\$\ \mathsf{HIBE.Enc}(pp, \vec{I}, \ell, m)\right] .$$

The probability is defined over the random coins used by HIBE.Set and HIBE.Enc, respectively. Note that min-entropy of HIBE.Set does not depend on the output value K^*.

Correctness.

Correctness requires that if a message m is encrypted to identity $\vec{I}$ and K is an identity key for that identity then K will decrypt the ciphertext properly. This is formalized by the game HIBECORR shown in Fig. 2.21. In it the adversary is given the public parameters pp and identity key K. It can make calls to $\textsc{Delegate}$ with a string I to ask for K to be updated to the identity $\vec{I} = \vec{I} \,\|\, I$. It will be given the new K. Finally it can query a label ℓ and message m to $\textsc{Enc}$. The message and label are then encrypted to the current identity $\vec{I}$. The produced ciphertext is immediately decrypted by the corresponding identity key K. If the decryption does not return the message that was encrypted then bad is set true. The adversary wins if it can cause bad to be set true.

Game $\text{HIBECCA}_{\mathsf{HIBE}}^{\mathcal{A}_{\mathsf{HIBE}}}$	$\text{Delegate}(\Upsilon,\Upsilon',I)$ / $\Upsilon,\Upsilon',I \in \{0,1\}^*$
$b \leftarrow\$ \{0,1\}$; $S \leftarrow \emptyset$; $S_I \leftarrow \emptyset$	If $K[\Upsilon] = \bot$ then return $\bot$
$(pp, K[0]) \leftarrow\$ \mathsf{HIBE.Set}$; $\mathbf{idtable}[0] \leftarrow ()$	If $K[\Upsilon'] \neq \bot$ then return $\bot$
$b' \leftarrow\$ \mathcal{A}_{\mathsf{HIBE}}^{\text{Enc,Dec,Delegate,Exp}}(pp)$	$K[\Upsilon'] \leftarrow\$ \mathsf{HIBE.Del}(K[\Upsilon], I)$
Return $(b = b')$	$\mathbf{idtable}[\Upsilon'] \leftarrow \mathbf{idtable}[\Upsilon] \parallel I$
$\text{Enc}(\vec{I}, m_0, m_1, \ell)$ / $\vec{I} \in (\{0,1\}^*)^*, m_0, m_1, \ell \in \{0,1\}^*$	Return $\bot$
If $\lvert m_0 \rvert \neq \lvert m_1 \rvert$ then return $\bot$	$\text{Exp}(\Upsilon)$ / $\Upsilon \in \{0,1\}^*$
If $\exists \vec{I}' \in S_I$ s.t. $\vec{I}' \sqsubseteq \vec{I}$ then return $\bot$	If $K[\Upsilon] = \bot$ then return $\bot$
$c \leftarrow\$ \mathsf{HIBE.Enc}(pp, \vec{I}, \ell, m_b)$	If $\exists (\vec{I}, c, \ell) \in S$ s.t. $\mathbf{idtable}[\Upsilon] \sqsubseteq \vec{I}$ then
$S \leftarrow S \cup \{(\vec{I}, c, \ell)\}$	Return $\bot$
Return c	$S_I \leftarrow S_I \cup \{\mathbf{idtable}[\Upsilon]\}$
$\text{Dec}(\Upsilon, c, \ell)$ / $\Upsilon, c, \ell \in \{0,1\}^*$	Return $K[\Upsilon]$
If $K[\Upsilon] = \bot$ then return $\bot$	
$v \leftarrow (\mathbf{idtable}[\Upsilon], c, \ell)$	
If $\exists (\vec{I}', c', \ell') \in S$ s.t. $(\vec{I}', c', \ell') = v$ then return $\bot$	
$m \leftarrow \mathsf{HIBE.Dec}(K[\Upsilon], \ell, c)$	
Return m	

Figure 2.22. Games defining CCA security of hierarchical public-key encryption scheme HIBE.

Algorithm PKE.Kg	Algorithm $\mathsf{PKE.Dec}(dk, \ell, c)$
$(pp, K) \leftarrow\$ \mathsf{HIBE.Set}$; $ek \leftarrow (pp, ())$	$m \leftarrow \mathsf{HIBE.Dec}(dk, \ell, c)$
Return (ek, K)	Return (dk, m)
Algorithm $\mathsf{PKE.Enc}(ek, \ell, m)$	Algorithm $\mathsf{PKE.UpdDk}(dk, \Delta)$
$(pp, \vec{I}) \leftarrow ek$; $c \leftarrow\$ \mathsf{HIBE.Enc}(pp, \vec{I}, \ell, m)$	$dk \leftarrow\$ \mathsf{HIBE.Del}(dk, \Delta)$
Return c	Return dk
Algorithm $\mathsf{PKE.UpdEk}(ek, \Delta)$	
$(pp, \vec{I}) \leftarrow ek$; $ek \leftarrow (pp, \vec{I} \parallel \Delta)$	
Return ek	

Figure 2.23. Key-updatable public-key encryption scheme $\mathsf{PKE} = \mathsf{PKE\text{-}CONS}[\mathsf{HIBE}]$.

The advantage of adversary $\mathcal{C}$ is defined by $\mathsf{Adv}_{\mathsf{HIBE}}^{\mathsf{hibecorr}}(\mathcal{C}) = \Pr[\text{HIBECORR}_{\mathsf{HIBE}}^{\mathcal{C}}]$. Perfect correctness requires that $\mathsf{Adv}_{\mathsf{HIBE}}^{\mathsf{hibecorr}}(\mathcal{C}) = 0$ for all (even unbounded) adversaries.

IND-CCA security of HIBE.

For security we will require that CCA security hold even when the adversary is given identity keys for arbitrarily many identities, as long as none of these identities are prefixes of

any identities it made a challenge query to. Security is formally defined by the game HIBECCA shown in Fig. 2.22.

In this game an adversary $\mathcal{A}_{\mathsf{HIBE}}$ is given the public parameters pp and then has access to five oracles. The initial identity key with identity $()$ is stored in $K[0]$. The adversary can ask for a sub-key to be delegated from an existing identity key $K[\Upsilon]$ with identity $\vec{I}$ by calling $\textsc{Delegate}$ with Υ, an unused key identifier Υ', and a string I. This oracle will create a new identity key $K[\Upsilon']$ with identity $\vec{I} \| I$. The adversary can ask for the encryption of a challenge message using $\textsc{Enc}$ to any identity $\vec{I}$ by calling it with input $\vec{I}$, m_0, m_1, and ℓ as long as no identity keys have been exposed for an identity that is a prefix of $\vec{I}$. The adversary can ask $\textsc{Dec}$ for the decryption of any (c, ℓ) pair by any existing identity key $K[\Upsilon]$ as long as the pair was not the output of a prior encryption query to the identity of $K[\Upsilon]$. Finally, the adversary can use $\textsc{Exp}$ to ask for the value of any identity key $K[\Upsilon]$ (which is associated with some identity $\vec{I}$) as long as it has not asked a challenge encryption query to an identity which $\vec{I}$ is a prefix of. The goal of the adversary is to guess the secret bit corresponding to which message $\textsc{Enc}$ encrypts.

Adversary $\mathcal{A}_{\mathsf{HIBE}}^{\textsc{Enc},\textsc{Dec},\textsc{Delegate},\textsc{Exp}}(pp)$	$\textsc{EncS}(\Lambda, m_0, m_1, \ell)$
$b' \leftarrow_{\$} \mathcal{B}_{\mathsf{PKE}}^{\textsc{NewUserS},\textsc{UpdEkS},\textsc{UpdDkS},\textsc{EncS},\textsc{DecS},\textsc{ExpS}}$ Return b'	If $\lvert m_0 \rvert \neq \lvert m_1 \rvert$ then return $\perp$ If $\vec{\Delta}'[\Lambda] \sqsubseteq \vec{\Delta}_e[\Lambda]$ then return $\perp$ $c \leftarrow \textsc{Enc}(\vec{\Delta}_e[\Lambda], m_0, m_1, \ell)$ $S[\Lambda] \leftarrow S[\Lambda] \cup \{(\vec{\Delta}_e[\Lambda], c, \ell)\}$ Return c
$\textsc{NewUserS}(\Lambda)$	$\textsc{DecS}(\Lambda, c, \ell)$
$\vec{\Delta}_e[\Lambda] \leftarrow ()\,;\ \vec{\Delta}_d[\Lambda] \leftarrow ()$ $\vec{\Delta}'[\Lambda] \leftarrow \perp\,;\ S[\Lambda] \leftarrow \emptyset$ $ek[\Lambda] \leftarrow (pp, \vec{\Delta}_e[\Lambda])$ Return $ek[\Lambda]$	If $(\vec{\Delta}_d[\Lambda], c, \ell) \in S[\Lambda]$ then return $\perp$ $m \leftarrow \textsc{Dec}(\lvert\vec{\Delta}_d[\Lambda]\rvert, c, \ell)$ Return m
$\textsc{UpdEkS}(\Lambda, \Delta)$	$\textsc{ExpS}(\Lambda)$
$\vec{\Delta}_e[\Lambda] \leftarrow \vec{\Delta}_e[\Lambda] \,\|\, \Delta$	If $\exists(\vec{\Delta}, c, \ell) \in S[\Lambda]$ s.t. $\vec{\Delta}_d[\Lambda] \sqsubseteq \vec{\Delta}$ then return $\perp$ If $\vec{\Delta}'[\Lambda] = \perp$ then $\vec{\Delta}'[\Lambda] \leftarrow \vec{\Delta}_d[\Lambda]$ Return $\textsc{Exp}(\lvert\vec{\Delta}_d[\Lambda]\rvert)$
$\textsc{UpdDkS}(\Lambda, \Delta)$	
$l \leftarrow \lvert\vec{\Delta}_d[\Lambda]\rvert\,;\ \textsc{Delegate}(l, l+1, \Delta)$ $\vec{\Delta}_d[\Lambda] \leftarrow \vec{\Delta}_d[\Lambda] \,\|\, \Delta$	

Figure 2.24. Adversary $\mathcal{A}_{\mathsf{HIBE}}$ used for proof of Theorem 12.

To keep track of the various things $\mathcal{A}_{\mathsf{HIBE}}$ is not allowed to do the game keeps a table **idtable** which maps key identifiers to the identity of that key, set S which stores all of the tuples $(\vec{I}, c, \ell)$ where (c, ℓ) was the output of an encryption query to identity $\vec{I}$, and set S_I which stores the identities of all keys that have been exposed. The advantage of adversary $\mathcal{A}_{\mathsf{HIBE}}$ is defined by $\mathsf{Adv}^{\mathsf{hibecca}}_{\mathsf{HIBE}}(\mathcal{A}_{\mathsf{HIBE}}) = 2\Pr[\mathrm{HIBECCA}^{\mathcal{A}_{\mathsf{HIBE}}}_{\mathsf{HIBE}}] - 1$.

Key-updatable public-key encryption scheme $\mathsf{PKE\text{-}CONS[HIBE]}$**.**

Let HIBE be a hierarchical identity based encryption scheme. We build a key-updatable public-key encryption scheme $\mathsf{PKE\text{-}CONS[HIBE]}$ as defined in Fig. 2.23. It essentially corresponds to using the HIBE directly by setting $ek = (pp, \vec{I})$ and $dk = K$. It is clear that $\mathrm{H}_\infty(\mathsf{PKE.Kg}) = \mathrm{H}_\infty(\mathsf{HIBE.Set})$ and $\mathrm{H}_\infty(\mathsf{PKE.Enc}) = \mathrm{H}_\infty(\mathsf{HIBE.Enc})$.

The following theorem bounds the advantage of an adversary performing an attack against $\mathsf{PKE\text{-}CONS[HIBE]}$ by the advantage of a similarly efficient adversary in attacking the security of HIBE. The theorem is relatively straightforward because the security of a key-updatable public key encryption scheme is essentially a special case of HIBE security.

Theorem 12. *Let* HIBE *be a hierarchical identity based encryption scheme and let* PKE *denote* $\mathsf{PKE\text{-}CONS[HIBE]}$. *Let* $\mathcal{A}_{\mathsf{PKE}}$ *be an adversary making at most* 1 *query to its* NEWUSER *oracle,* q_{UPDEK} *queries to its* UPDEK *oracle,* q_{UPDDK} *queries to its* UPDDK *oracle,* q_{ENC} *queries to its* ENC *oracle,* q_{DEC} *queries to its* DEC *oracle, and* q_{EXP} *queries to its* EXP *oracle. Then we can build adversary* $\mathcal{A}_{\mathsf{HIBE}}$ *against the security of* HIBE *such that*

$$\mathsf{Adv}^{\mathsf{indexp}}_{\mathsf{PKE}}(\mathcal{A}_{\mathsf{PKE}}) \leq \mathsf{Adv}^{\mathsf{hibecca}}_{\mathsf{HIBE}}(\mathcal{A}_{\mathsf{HIBE}}). \tag{2.3}$$

Adversary $\mathcal{A}_{\mathsf{HIBE}}$ *makes at most* q_{ENC} *queries to its* ENC *oracle,* q_{DEC} *queries to its* DEC *oracle,* q_{UPDDK} *queries to its* DELEGATE *oracle, and* q_{EXP} *queries to its* EXP *oracle. The running time of* $\mathcal{A}_{\mathsf{HIBE}}$ *is about that of* $\mathcal{A}_{\mathsf{PKE}}$.

For simplicity we show that single-user security of PKE (i.e. $\mathcal{A}_{\mathsf{PKE}}$ makes at most 1 query to is NEWUSER oracle) is obtained from the single-user security of HIBE. This tight

reduction between two single-user security notions can be generically transformed into a tight reduction between the corresponding multi-user notions.

Theorem 12. Because $\mathcal{A}_{\mathsf{PKE}}$ makes at most one query to NEWUSER, we will without loss of generality consider the adversary an $\mathcal{B}_{\mathsf{PKE}}$ which makes its first query to NEWUSER and makes all queries with a single, fixed value of Λ because there must exist such an adversary with efficiency about that of $\mathcal{A}_{\mathsf{PKE}}$ which satisfies $\mathsf{Adv}^{\mathsf{indexp}}_{\mathsf{PKE}}(\mathcal{A}_{\mathsf{PKE}}) \leq \mathsf{Adv}^{\mathsf{indexp}}_{\mathsf{PKE}}(\mathcal{B}_{\mathsf{PKE}})$.

Now consider the adversary $\mathcal{A}_{\mathsf{HIBE}}$ shown in Fig. 2.24. It simulates the view of $\mathcal{A}_{\mathsf{PKE}}$ in the obvious way using its HIBECCA oracles. Its code was obtained by plugging the code of PKE into INDEXP and then, where appropriate, replacing executing algorithms of HIBE with oracle queries by $\mathcal{A}_{\mathsf{HIBE}}$ and removing code which is irrelevant for $\mathcal{B}_{\mathsf{PKE}}$.

The underlying secret bit of HIBECCA plays the role of the secret bit in INDEXP. We need to argue that the simulated view of $\mathcal{A}_{\mathsf{PKE}}$ is identical to its view in INDEXP. This would be immediate to verify if the various oracles in HIBECCA always returned their intended value; however, sometimes the abort early, returning $\perp$. We will analyze the possible ways of this occurring individually.

For all of the oracles in HIBECCA other than have checks about whether entries of $K[\Upsilon]$ have or have not been initialized yet. Note that $\mathcal{A}_{\mathsf{HIBE}}$ uses the lengths of $\vec{\Delta}_d[\Lambda]$. This is incremented by 1 with each UPDDKS query. Based on this we can verify that these checks will never cause the oracles of HIBECCA to abort early when called by $\mathcal{A}_{\mathsf{HIBE}}$.

Adversary $\mathcal{A}_{\mathsf{HIBE}}$ will never query ENC with m_0 and m_1 of different length, so the corresponding check in ENC will never cause it to abort early.

The remaining checks we must analyze are those that depend on the sets S and S_I. Note that for any $\vec{I} \in S_I$ it will hold that $\vec{\Delta}'[\Lambda] \sqsubseteq \vec{I}$. Similarly for any $(\vec{I}, c, \ell) \in S$ it will hold that $(\vec{I}, c, \ell) S[\Lambda]$. From this it is clear that the checks $\mathcal{A}_{\mathsf{HIBE}}$ performs before making oracle queries will prevent any of its oracles from returning $\perp$ because of the checks depending on these sets.

Hence it is clear that $\mathcal{A}_{\mathsf{HIBE}}$ correctly guesses the secret bit whenever $\mathcal{B}_{\mathsf{PKE}}$ would guess

its secret bit, so $\mathsf{Adv}^{\mathsf{indexp}}_{\mathsf{PKE}}(\mathcal{B}_{\mathsf{PKE}}) \leq \mathsf{Adv}^{\mathsf{hibecca}}_{\mathsf{HIBE}}(\mathcal{A}_{\mathsf{HIBE}})$, completing the proof. $\square$

2.10 Acknowledgements

We thank Mihir Bellare for extensive discussion on preliminary versions of this work. We thank the CRYPTO 2018 reviewers for their comments.

Chapter 2, in full, is a reprint of the material as it appears in Advances in Cryptology - CRYPTO 2018. Jaeger, Joseph; Stepanovs, Igors, Springer Lecture Notes in Computer Science volume 10991, 2018. The dissertation author was the primary investigator and author of this paper. The dissertation author was supported in part by NSF grants CNS-1717640 and CNS-1526801.

Chapter 3

Key Exchange for Messaging Apps

In this work we study key exchange protocols used to initiate sessions in end-to-end encrypted messaging apps, aiming to provide a security definition which is efficiently achievable yet strong enough to maintain security when composed with arbitrary messaging protocols.

Why key exchange?

A key observation motivating the design of Signal is that users often use messaging apps over long periods of time during which all of their cryptographic secrets will be stored on their personal devices and possibly vulnerable to being leaked by malware or attackers having physical access to their device. Signal attempts to address this issue by updating cryptographic secrets over time. That way communication which occurs before or after the device compromise may still be secured.

The first academic analysis of Signal was done by Cohn-Gordan, Cremers, Dowling, Garratt, and Stebila [29] who studied the entirety of its key exchange. This includes the initial exchange of keys to start a conversation and the later continuous updates of keys over time during a conversation.

In Chapter 1 we proposed that this problem should be studied in a more modular fashion — separating the initial exchange of cryptographic secrets from the later use during which these secrets are updated. In both previous chapters we (and others in a number of followup works [56, 35, 45, 2]) focus on the post-key exchange use of the secrets. The initial exchange of

secrets is abstracted away as an “initialization” algorithm which outputs the initial states of two parties in a trusted manner. These works leave unspecified how this initialization can be done in practice by two parties communicating over an untrusted channel. Our works aims to bridge that gap by defining and analyzing a key exchange model appropriate for this setting. One of the primary contributions of our work is a composition result showing that key exchanges secure in our model compose correctly with using the keys for the messaging protocols described in these numerous prior works. Below we specify a number of important aspects of this setting which differ from traditional key exchange models.

We refer to the setting as “offline server-aided public and private key exchange” using “key fingerprints” to provide authenticity and achieving strong security against “state compromise”. Let us dig into what is mean by each part of this phrase.

Offline server-aided.

In traditional key exchange both parties are assumed to be online during the execution, so they can directly exchange messages with each other. This does not capture the reality of messaging applications where a user may wish to send an initial message to their friend even though their friend is not on their phone at the moment. What happens in practice is that messaging app provider makes available a server. All users of the app register some information (e.g. public keys) with the server. Then later when a user wants to initiate a new chatting session with their friend they talk to the server. The server provides them the necessary information to finish their half of the key exchange and thereby encrypt messages to their friend. Later when the friend is online they can interact with the server to finalize the key exchange - obtaining the keys they need to receive the already sent messages from the user and send messages back.

Note that multiple users may try to initiate sessions with the friend between when they register and are next online again. The registration should then allow the server to start multiple key exchanges on their behalf. Optimally we would prefer that a constant amount of state on the server allowed for an arbitrary number of key exchanges; however, it seems unlikely that this

would be possible without knowing the structure of the keys being exchanged so in the generic scheme we define the state is linear in the number of key exchanges that can be started.

Our formalism considers a key exchange which consists of multiple stages: a registration phase, a request phase, and a finalization phase.

Public and private key exchange.

Typically key exchange definitions deal with exchange of a single symmetric key which will be shared between the two communicating parties. However, the keys required for our setting tend to be more structured than that. As an example, the keys used by the Signal messaging protocol (in addition to a shared symmetric key) consist of Diffie-Hellman pairs. For each pair one of the two users will hold the secret value while the other user holds only the public value. The initial key exchange of Signal must also exchange the appropriate Diffie-Hellman values.

One could use a typical key exchange and then derive the Diffie-Hellman values from the produced key, but then both parties will know the secret part of the pair. This would potentially weaken the security guarantees provided by the messaging protocol when using these values.

Our formalism captures the exchange of more structured keys. This requires care in defining our security notions for key exchange. As an example, key indistinguishability notions in the vein of Bellare-Rogaway models will not be applicable because the public parts of the keys exchanged will make the secret parts no longer indistinguishable from random.

Key fingerprinting.

An important notion in key exchange security is that of authentication. Users want to be assured that they exchanged keys with the person they intended to. In TLS, for example, these sorts of guarantees are provided by the use of certificates. In messaging applications the more typical technique is to use "key fingerprints". This consists of both users' devices being made to compare some short string. Authenticity of the key exchange is assured as long as these string match. This comparison may be done by sharing the string out-of-band over some trusted medium (e.g. reading the string out over a telephone or scanning a QR code on one person's

device with the other person's device). To match this, our formalism includes a model of key fingerprinting to allow authenticity. This is related to Vaudenay's model of achieving authenticity based on "Short Authenticated Strings" [65].

Security against state compromise.

In modeling key exchange it is common to account for the fact that some users' secret state may be compromised by the adversary, allowing them to attack key exchange sessions those users are involved in. Security definitions then typically require that the key exchange sessions of all *other* users are still secure. Additionally, they may require *forward security*, i.e., that all sessions prior to the state compromise remained secure.

In our setting we can improve on this default in two directions. The first is to provide *post-compromise* security. In our formalism we allow a user to repeatedly refresh their state by interacting with the server to re-register what information is stored there. The desired guarantee is that a theft of state between two refreshes only compromises the security of key exchanges initiated between these refreshes.

Towards the second direction of improvement, note that the key exchange we study provides keys to a messaging protocol which itself may provide security against state compromise. The latter is typically provided by the messaging protocol continuously updating its secrets over time. We take care in defining our model so that state compromise during the key exchange corresponds directly to a compromise of the initial state in the later messaging protocol. Thus whatever mechanisms are used by the messaging protocol to mitigate against state compromise can serve to mitigate against compromises that happened during the key exchange. Some of the issues we identify while proving this may serve to provide guidance for how future security definitions for messaging protocols should be written.

Key exchange composability.

There are several ways the composability of a key exchange protocol may be proven. A first approach is to simply define its security in a simulation based framework such as UC

security [24, 27] for which a generic composition result has already been proven. However, these frameworks typically place very stringent restrictions which make achieving security difficult and moreover the generic composition results typically apply specifically to achieving other security notions in the same framework. A second approach would be to directly prove composition for a specific underlying goal such as the work of Canetti and Krawczyk [26] which showed a composition result specifically for building secure channel.

While our OKE security definition is simulation based in nature, this work is perhaps most spiritually similar to that of Brzuska, Fischlin, Warinschi, and Williams [23]. In that work they provide an abstract framework of game-based definitions for symmetric key protocols and what it means to compose a key exchange protocol with a symmetric key protocol. Then they prove that any key exchange protocol meeting a Bellare-Rogaway style security definition maintains security when composed with a symmetric key protocol. Our work mirrors this. First we provide a framework of game-based security definitions for messaging protocols which provide resilience against state compromise and define what it means to compose an OKE protocol with such a messaging protocol. Then we introduce a simulation based security notion for OKE and show that secure OKE protocols compose correctly with messaging protocols. Our use of a simulation style comes not from an aesthetic preference, but instead from the fact that there doesn't appear to be any way to modify a Bellare-Rogaway style indistinguishability notion to fit our setting.

Other related work.

Cohn-Gordon, Cremers and Garratt [28] studied post-compromise security for authenticated key exchange. However their setting is distinct from ours. Most importantly they achieve post-compromise security for repeated key exchanges between the *same* conversation partners by storing state between them; our post-compromise security is between sessions with different partners.

Shoup [61] introduced a simulation style security notion which is claimed (but not proven)

to have good composability properties.

3.1 Notation and conventions

We use the code-based game playing framework of [16]. By Pr[G] we denote the event that the execution of game G results in the game returning true. "Require bool" is shorthand for the pseudocode "If not bool then return $\perp$".

A list T is an ordered list of entries. Object x can be added to T via $T.\mathsf{add}(x)$. The first element added to T is $T[1]$, the second is $T[2]$, etc. We sometimes interpret T as the set $\{T[i] : T[i] \neq \perp\}$. For example the intersection of this set and a set X is denoted $T \cap X$. The code "For $x \in T$" traverses T in FIFO order. $T.\mathsf{pop}()$ removes and returns the last element added to the list. We use similar notation for a table T. The operation $T[i] \leftarrow x$ denotes storing x in T at location i where i may be an arbitrary string.

If X is a finite set, we let $x \leftarrow\!\!\$\, X$ denote picking an element of X uniformly at random and assigning it to x. Algorithms may be randomized unless otherwise indicated. If A is an algorithm, we let $y \leftarrow\!\!\$\, A(x_1, \ldots)$ denote running A with fresh random coins and assigning its output to y. Variable on the left sign of an assignment are notated by $\cdot$ when their value will not be used. For example $(x, \cdot) \leftarrow y$ denotes parsing y as a tuple and assigning x to the first value in the tuple. The other value is ignored.

3.2 Messaging Security

Our goal is to understand how to design key exchange protocols which compose correctly with messaging protocols that provide some sort of security against exposure of their secrets. Such messaging protocols have been studied in a number of recent works [17, 56, 44, 35, 45, 2]. Each of these has their own specific definition of what a messaging protocol is and what security is expected of it.

It is infeasible to try to analyze key exchanges protocols with respect to each of these

individually. Instead we introduce a general framework of definitions. This framework captures a minimal specification of what a messaging scheme and corresponding notion of security against state exposure look like. This general framework captures all of these existing schemes and security definitions (up to some small modification we will discuss later). Additionally, the framework should serve to capture definitions used by any future additions to this line of work.

The generality of the framework is obtained by making it as sparse as possible; it assumes only the minimal structure of messaging schemes and security definitions to allow our analysis. Of messaging schemes we assume only that they include an "initialization" algorithm which produces state for two parties. Of the security definitions we assume that they are game-based and that one of the oracles via which the attacker can interact with the messaging scheme is an exposure oracle EXP which returns the current secret state of one of the two parties.

Messaging scheme.

We assume the bare minimum syntax for a messaging protocol. They consider communication between two parties: an initiator $\mathcal{I}$ and a responder $\mathcal{R}$. Formally a messaging scheme Π is a collection of algorithms, one of which is an initialization algorithm $\Pi.\mathsf{Init}$ which produces initial keys for two communicating parties. This has the form $(\ell, \vec{k}_{\mathcal{I}}, \vec{k}_{\mathcal{R}}) \leftarrow\$ \Pi.\mathsf{Init}$. Here ℓ is information about the initial keys which is not presumed to be secret (e.g. public keys), $\vec{k}_{\mathcal{I}}$ is the initial state created for the party $\mathcal{I}$, and $\vec{k}_{\mathcal{R}}$ is the initial state created for the party $\mathcal{R}$.

A messaging scheme will typically also specify a "sending" algorithm and a "receiving" algorithm. These algorithms use the states of the parties to send information between them. In the process they potentially update these states. For our purposes we do not need to know the specifics of the syntax of these algorithms.

It will later be convenient to limit our attention to initialization algorithm which are *splittable*. A splittable initialization algorithm is specified by algorithms $\Pi.\mathsf{Kg}_{\mathcal{I}}$ and $\Pi.\mathsf{Kg}_{\mathcal{R}}$ together with keyspace $\{0,1\}^{\Pi.\mathsf{k}}$. The algorithms allow $\mathcal{I}$ and $\mathcal{R}$ to separately generate their own pairs of public and secret keys. A shared secret key is sampled from $\{0,1\}^{\Pi.\mathsf{k}}$. The splittable

$\Pi.\mathsf{Init}$ algorithm behaves as follows. We typically think of $k_{\mathcal{R}}$ as having been sampled by $\mathcal{R}$.

$\underline{\Pi.\mathsf{Init}}$

$(pk_{\mathcal{I}}, sk_{\mathcal{I}}) \leftarrow\$ \Pi.\mathsf{Kg}_{\mathcal{I}}$; $(pk_{\mathcal{R}}, sk_{\mathcal{R}}) \leftarrow\$ \Pi.\mathsf{Kg}_{\mathcal{R}}$; $k_{\mathcal{R}} \leftarrow\$ \{0,1\}^{\Pi.\mathsf{k}}$

$\ell \leftarrow (pk_{\mathcal{I}}, pk_{\mathcal{R}})$; $\vec{k}_{\mathcal{I}} \leftarrow (\ell, sk_{\mathcal{I}}, k_{\mathcal{R}})$; $\vec{k}_{\mathcal{R}} \leftarrow (\ell, sk_{\mathcal{R}}, k_{\mathcal{R}})$

Return $(\ell, \vec{k}_{\mathcal{I}}, \vec{k}_{\mathcal{R}})$

Our notion of splittable initialization algorithms is a special case of the notion as defined by Durak and Vaudenay [35] which does not assume that all of ℓ and $k_{\mathcal{R}}$ are given to both parties.

State exposure games.

The security notions of a messaging scheme Π are captured by security games which allow the attacker to interact with Π. Consider the game $\mathrm{G}^{\mathcal{G}}_{\Pi,b}$ shown below. It is parameterized by a *game specification* $\mathcal{G}$, a messaging scheme Π, and a bit b that the attacker attempts to guess. In it, $\Pi.\mathsf{Init}$ is run to produce initial states for $\mathcal{I}$ and $\mathcal{R}$. The adversary $\mathcal{A}$ is given the leakage ℓ together with access to oracles O and EXP. Finally the attacker outputs a bit b' representing its guess of b. The advantage of $\mathcal{A}$ is then defined by $\mathsf{Adv}^{\mathcal{G}}_{\Pi}(\mathcal{A}) = \Pr[\mathrm{G}^{\mathcal{G}}_{\Pi,1}] - \Pr[\mathrm{G}^{\mathcal{G}}_{\Pi,0}]$.

Game $\mathrm{G}^{\mathcal{G}}_{\Pi,b}(\mathcal{A})$	$\underline{\mathrm{EXP}(\mathsf{lab})}$
$(\ell, \vec{k}[\mathcal{I}], \vec{k}[\mathcal{R}]]) \leftarrow\$ \Pi.\mathsf{Kg}$ $\sigma_{\mathcal{G}} \leftarrow\$ \mathcal{G}.\mathbf{Start}(b)$ $b' \leftarrow\$ \mathcal{A}^{\mathrm{O,EXP}}(\ell)$ If $\mathcal{G}.\mathbf{Pred}(\mathbf{T})$ then $b' \leftarrow 0$ Return $(b' = 1)$	$\mathbf{T}.\mathsf{add}(\mathrm{EXP}, \mathsf{lab}, \vec{k}[\mathsf{lab}])$ Return $\vec{k}[\mathsf{lab}]$
$\underline{\mathrm{O}(x)}$ $(y, \vec{k}[\mathcal{I}], \vec{k}[\mathcal{R}], \sigma_{\mathcal{G}}) \leftarrow\$ \mathcal{G}^{\Pi}(\sigma_{\mathcal{G}}, x, \vec{k}[\mathcal{I}], \vec{k}[\mathcal{R}])$ $\mathbf{T}.\mathsf{add}(\mathrm{O}, x, y)$ Return y	

The exposure oracle EXP allows the attacker to obtain the current secret state of either of the parties. The oracle O captures any other ways that the attacker may be allowed to interact with the scheme Π. (To capture the security games of most of the previous works we are interested in, one simply has O multiplex all of the additional oracles the adversary would have

been given access to.) Note that interacting with Π can result in the state of the parties being updated over time.

The variable $\mathbf{T}$ is a *transcript* used for tracking the oracle queries of the adversary. Each time an oracle query is made it stores which oracle, what the input to the oracle was, and what value was returned to $\mathcal{A}$.

To prevent trivial attacks security games typically need to place some restrictions on the behavior of $\mathcal{A}$. For example in an encryption context an attacker that exposes the state of $\mathcal{R}$ can necessarily decrypt the next message $\mathcal{R}$ would receive, so $\mathcal{A}$ should be prevented from trivially using this ability to learn the bit b. These restrictions are facilitated by the predicate $\mathcal{G}.\mathbf{Pred}(\mathbf{T})$. The predicate checks the final transcript to see if $\mathcal{A}$ disobeyed the restrictions; if so the predicate returns true which sets b' to 0 regardless of what $\mathcal{A}$ actually output. We sometimes refer to this as rejecting the transcript. In our example, $\mathcal{G}.\mathbf{Pred}(\mathbf{T})$ could return true if the transcript shows that a state of $\mathcal{R}$ was exposed which would allow $\mathcal{A}$ to decrypt a message whose value depended on b.

Guidance for future security definition.

We identify two aspects of our security definition which are essential to our work and should be captured in all future security games for messaging protocol. These are that attention should be restricted to splittable initialization algorithms and that the public components of these algorithms should be provided to the adversary at the start of the game. Of the various security definitions we considered [17, 56, 44, 35, 45, 2] only Durak and Vaudenay [35] restricted attention to splittable initialization algorithms. Additionally they and Bellare, et. al. [17] were the only ones to reveal the public components to the attacker. However it seems to be the case that *all* of the schemes happen to have splittable initialization algorithms and would provide security if the initial public components were leaked so our later composition result can still be thought to apply to them.

A further idea we note later in Section 3.3.2 is that it is not possible to provide generic

composition results for OKE protocols based on key-encapsulation methods (KEMs) for generic state exposures games as we have defined them. To allow more efficient protocols based on KEMs future security definitions for messaging protocols may consider additionally allowing the adversary to arbitrarily *choose* $k_{\mathcal{R}}$ if it exposes the initial state of either user.

3.3 Key exchange

Exchange of private and public keys.

Most existing work on key exchange is concerned solely with the exchange of a shared secret symmetric key. For this work we need to understand how to define security for a key exchange that shares asymmetric keys. In particular, we want a key exchange that instantiates running a splittable initialization algorithm and then giving $\vec{k}_{\mathcal{I}}$ to one party and $\vec{k}_{\mathcal{R}}$ to the other while leaking nothing beyond ℓ.

Server-aided offline key-exchange.

The setting of server-aided offline key-exchange (OKE for short) involves a single server $\mathcal{S}$ and any number of users.

An OKE protocol KE specifies the following subprotocols. Here the notation $(y_a, y_b) \leftarrow\$$ $\mathsf{P}(x_a : x_b)$ indicates an interaction between two parties with local inputs x_a and x_b that produces local outputs y_a and y_b. Sometimes state variables will appear as both input and output of a protocol, this represents the state being updated during the execution of the protocol. When the server interacts with a user u it is always given u as input. This is intended to model the server having first authenticated the identity of the user before initiation of this protocol.

- KE.SKg: The server key generation algorithm which produces the initial state of $\mathcal{S}$ via $st_{\mathcal{S}} \leftarrow\$ \mathsf{KE.SKg}$.
- KE.PKg: The party key generation algorithm which produces the initial state of party u via $st_u \leftarrow\$ \mathsf{KE.PKg}(u)$.
- KE.Reg: The registration protocol in which a user u initially provides information about

itself to the server $\mathcal{S}$. We write $(st_u, st_{\mathcal{S}}) \leftarrow\$ \mathsf{KE.Reg}(st_u : (st_{\mathcal{S}}, u))$. A user may later re-register (by running this protocol again) to update the information stored by the server.

- $\mathsf{KE.Req}$: The requesting protocol in which a (session of a) user u requests a connection to another user v. The session finishes its role in the key exchange and outputs the keys it has derived. We write $((st_u, \vec{k}_u), st_{\mathcal{S}}) \leftarrow\$ \mathsf{KE.Req}((st_u, v) : (st_{\mathcal{S}}, u))$. If something goes wrong, u rejects by setting $\vec{k}_u = \bot$.
- $\mathsf{KE.Fin}$: The finalization protocol in which a (session of a) user v completes a key exchange with u. We write $((st_v, \vec{k}_v), st_{\mathcal{S}}) \leftarrow\$ \mathsf{KE.Fin}((st_v, u) : (st_{\mathcal{S}}, v))$. If something goes wrong, v rejects by setting $\vec{k}_v = \bot$.

We think of $\mathsf{KE.Reg}$ as corresponding to user $\mathcal{I}$ running $\Pi.\mathsf{Kg}_{\mathcal{I}}$ to generate pairs of public and secrets keys $(pk_{\mathcal{I}}, sk_{\mathcal{I}})$. Then we think of $\mathsf{KE.Req}$ as another user $\mathcal{R}$ receiving $pk_{\mathcal{I}}$, running $\Pi.\mathsf{Kg}_{\mathcal{R}}$ to generate pairs of public and secrets keys $(pk_{\mathcal{R}}, sk_{\mathcal{R}})$, and sampling $k_{\mathcal{R}}$ from $\{0,1\}^{\Pi.\mathsf{k}}$. Then finally $\mathsf{KE.Fin}$ corresponds to $\mathcal{I}$ receiving $pk_{\mathcal{R}}$ and $k_{\mathcal{R}}$. For notational simplicity we denote the algorithms that KE tries to emulate by $\mathsf{KE.Init}$, $\mathsf{KE.Kg}_{\mathcal{I}}$, $\mathsf{KE.Kg}_{\mathcal{R}}$, and $\{0,1\}^{\mathsf{KE.k}}$.

Canonical form of OKE.

For defining our security models we will notationally simplify things by only considering KE protocols that fall within a particular canonical form, represented via pseudocode in Fig. 3.1. This essentially corresponds to assuming that each of the protocols above uses the minimal required amount of interaction. We will assume that registration consists of the user sending a message ($\mathsf{KE.UReg}$) to the server with no response back required ($\mathsf{KE.SReg}$). We assume the requesting protocol consists of the server sending a single message to the user ($\mathsf{KE.SReq}_1$) and the user sending a single message back ($\mathsf{KE.UReq}$) to be processed by the server ($\mathsf{KE.SReq}_2$). The user will then wait for a fingerprint comparison to decide if it will accept the session. Finalization will consist of the server sending a single message ($\mathsf{KE.SFin}$) to the user ($\mathsf{KE.UFin}$). Then the user waits for a fingerprint comparison to decide if it will accept the session. It is assumed that $\mathsf{KE.UReq}$ and $\mathsf{KE.UFin}$ output $fp = \bot$ if and only if they output $\vec{k} = \bot$.

$\underline{\mathsf{KE.Reg}(st_u : (st_{\mathcal{S}}, u))}$
$(st_u, m) \leftarrow\!\!\$\ \mathsf{KE.UReg}(st_u)$
u sends m to $\mathcal{S}$
u returns $\vec{k}_u$
$st_{\mathcal{S}} \leftarrow\!\!\$\ \mathsf{KE.SReg}(st_{\mathcal{S}}, m, u)$
$\mathcal{S}$ returns $st_{\mathcal{S}}$

$\underline{\mathsf{KE.Req}((st_v, u) : (st_{\mathcal{S}}, v))}$
v sends u to $\mathcal{S}$
$(st_{\mathcal{S}}, m) \leftarrow\!\!\$\ \mathsf{KE.SReq}_1(st_{\mathcal{S}}, v, u)$
$\mathcal{S}$ sends m to v
$(st_v, m, \vec{k}_v, fp)$
  $\leftarrow\!\!\$\ \mathsf{KE.UReq}(st_v, m)$
v sends m to $\mathcal{S}$
If $\vec{k}_v = \bot$ then u returns $(st_v, \bot)$
Else (while server proceeds)
  v waits for fp comparison
  If matches returns $(st_v, \vec{k}_v)$
  Otherwise returns $(st_v, \bot)$
$st_{\mathcal{S}} \leftarrow\!\!\$\ \mathsf{KE.SReq}_2(st_{\mathcal{S}}, m, v, u)$
$\mathcal{S}$ returns $st_{\mathcal{S}}$

$\underline{\mathsf{KE.Fin}((st_u, v) : (st_{\mathcal{S}}, u))}$
u sends v to $\mathcal{S}$
$(st_{\mathcal{S}}, m) \leftarrow\!\!\$\ \mathsf{KE.SFin}(st_{\mathcal{S}}, u, v)$
$\mathcal{S}$ sends m to u
$\mathcal{S}$ returns $st_{\mathcal{S}}$
$(st_u, \vec{k}_u, fp) \leftarrow\!\!\$\ \mathsf{KE.UFin}(st_u, m)$
If $\vec{k}_u = \bot$ then u returns $(st_u, \bot)$
Else
  u waits for fp comparison
  If matches returns $(st_u, \vec{k}_u)$
  Otherwise returns $(st_u, \bot)$

Figure 3.1. Canonical form of key exchange protocol.

3.3.1 Adversary Model

We now describe our adversary model. We start with an abstract textual description of how an attacker may interact with an OKE protocol. Then for different security notions we will fix concrete pseudocode instantiating these interactions. Recall that we think of the attacker as controlling the server $\mathcal{S}$. This puts the attacker in complete control of all communication with the different users. During this communication their behavior may arbitrarily diverge from the correct execution of the server as specified by KE. In typically key exchange fashion the attacker is also in charge of deciding which users attempt to exchange keys with which other users and at what time.

Notation.

Users of the key exchange will be identified by names $u \in \{0,1\}^*$. The server should store state $st_{\mathcal{S}}$ and a user u will store state st_u. Session i of user u will store session specific state sst_u^i. Each user may have an arbitrary number of sessions denoted by identifiers $i \in \{0,1\}^*$. Sessions are created by a user each time it participates in a $\mathsf{KE.Req}$ or $\mathsf{KE.Fin}$ protocol. Note that i will be an administrative label used by the security model, but unknown to the protocol. For session i of user u the model will store state in the variable sst_u^i. It will store the following values indexable by dot notation. For example, the sess stored by u's session i is the variable $sst_u^i.\mathsf{sess}$. We sometimes refer to u's session i as the session (u,i).

- $\mathsf{id} = u$: the name of the session owner
- $\mathsf{sess} = i$: the session identifier
- pid: the name of the intended partner user of this session
- psess: the session identifier of the partner session
- $\mathsf{status} \in \{\mathsf{waiting}, \mathsf{accepted}, \mathsf{rejected}\}$: the status of the session, $\mathsf{waiting}$ when waiting for a fingerprint comparison, $\mathsf{accepted}$ or $\mathsf{rejected}$ when the session is finished
- fing: the fingerprint to be compared with that of the partner session
- keys: the keys produced by this session
- $\mathsf{role} \in \{\mathcal{I}, \mathcal{R}\}$: the role of the session which is set to $\mathcal{R}$ in the $\mathsf{KE.Req}$ protocol or $\mathcal{I}$ in $\mathsf{KE.Fin}$

Interacting with OKE.

Recall that in our model the server *is* the adversary. As such, none of the server algorithms will be run by the game. All messages that would be sent to the server are given to the adversary and all messages that would be expected from the server are specified by the adversary. The adversary may, of course, choose to locally run some of the correct server algorithms while interacting with the users.

After initialization the adversary is given access to several oracles which allow it to interact with the key exchange protocol. In particular, there are oracles capturing the three ways that a user might interact with the server (registration, requesting, and finishing), an oracle for making two sessions do a fingerprint comparison, and two oracles for learning local or session state of users. We now describe them one at a time.

- REG: The adversary specifies a user u to register (or re-register) with the server. The registration messages is given to the adversary.

- REQ: The adversary creates a new session i for user u by sending it the server message from a request protocol execution. The specified session must not already exist. The session either rejects or has set a fingerprint and is waiting for a comparison to complete its exchange. The session's response message and fingerprint are given to the adversary.

- FIN: The adversary creates a new session i for user u by sending it the server message from a finalize protocol execution. The user must be registered and the specified session must not already exist. The session either rejects or has set a fingerprint and is waiting for a comparison to complete its exchange. The session's fingerprint it given to the adversary.

- FING: The adversary causes user u's session i and user v's session j to perform a fingerprint comparison. Both specified sessions must be waiting for a fingerprint comparison. If it succeeds they both accept and are set to be each other's partners. Otherwise they will both reject.

- CORRUPT: The adversary corrupts user u and is given its current local state.

- REVEAL: The adversary reveals the secret state of user u's session i.

Pseudocode for these different methods of interaction is given in Fig. 3.2 The fingerprints being returned to the adversary represents the fact that we do not want to place restrictions on how fingerprints are actually compared. For example, users can safely compare them by publicly posting them on their respective social media profiles.

$\underline{\text{REG}(u)}$	$\underline{\text{REQ}(u,i,m,v)}$
$(st_u, m) \leftarrow\$ \mathsf{KE.UReg}(st_u)$	$(st_u, m', \vec{k}_u, fp)$
$\mathbf{T}.\mathsf{add}(\text{REG}, u, m)$	$\quad \leftarrow\$ \mathsf{KE.UReq}(st_u, m)$
Return m	$sst_u^i.\mathsf{keys} \leftarrow \vec{k}_u$
$\underline{\text{FIN}(u,i,m,v)}$	$sst_u^i.\mathsf{fing} \leftarrow fp$
$(st_u, \vec{k}_u, fp) \leftarrow\$ \mathsf{KE.UFin}(st_u, m)$	$sst_u^i.\mathsf{role} \leftarrow \mathcal{R}$
$sst_u^i.\mathsf{keys} \leftarrow \vec{k}_u$	$sst_u^i.\mathsf{pid} \leftarrow v$
$sst_u^i.\mathsf{fing} \leftarrow fp$	If $\vec{k}_u = \bot$ then
$sst_u^i.\mathsf{role} \leftarrow \mathcal{I}$	$\quad sst_u^i.\mathsf{status} \leftarrow \mathsf{rejected}$
$sst_u^i.\mathsf{pid} \leftarrow v$	Else
If $\vec{k}_u = \bot$ then	$\quad sst_u^i.\mathsf{status} \leftarrow \mathsf{waiting}$
$\quad sst_u^i.\mathsf{status} \leftarrow \mathsf{rejected}$	$\mathbf{T}.\mathsf{add}(\text{REQ}, (u,i,m,v), (m', fp))$
Else	Return (m', fp)
$\quad sst_u^i.\mathsf{status} \leftarrow \mathsf{waiting}$	$\underline{\text{CORRUPT}(u)}$
$\mathbf{T}.\mathsf{add}(\text{FIN}, (u,i,m,v), fp)$	$\mathbf{T}.\mathsf{add}(\text{CORRUPT}, u, st_u)$
Return fp	Return st_u
	$\underline{\text{REVEAL}(u,i)}$
	$\mathbf{T}.\mathsf{add}(\text{REVEAL}, (u,i), sst_u^i.\mathsf{keys})$
	Return $sst_u^i.\mathsf{keys}$

Figure 3.2. Oracles specifying how the attacker can interact with OKE protocol KE.

Composed security.

The ultimate goal of an OKE scheme KE is to be able to securely initiate the keys of a messaging scheme Π that it is composed with. Recall that in Section 3.2 the security of Π is measured by $\mathrm{G}^{\mathcal{G}}_{\Pi,b}$ instantiated with a particular game specification $\mathcal{G}$. To understand the security of KE and Π together we then define a security game $\mathrm{G}^{\text{oke-}\mathcal{G}}_{\Pi,\mathsf{KE},\mathcal{P},b}$ which maps $\mathcal{G}$ to a multi-user variant of $\mathrm{G}^{\mathcal{G}}_{\Pi,b}$ in which keys are shared between users via KE instead of just handed out after an execution of Π.Init. This game is shown in Fig. 3.3 with additional oracles from Fig. 3.2. The advantage of an attack $\mathcal{A}$ playing this game is defined by $\mathsf{Adv}^{\text{oke-}\mathcal{G}}_{\Pi,\mathsf{KE},\mathcal{P}}(\mathcal{A}) = \Pr[\mathrm{G}^{\text{oke-}\mathcal{G}}_{\Pi,\mathsf{KE},\mathcal{P},1}(\mathcal{A})] - \Pr[\mathrm{G}^{\text{oke-}\mathcal{G}}_{\Pi,\mathsf{KE},\mathcal{P},0}(\mathcal{A})]$.

At a high level, the security notion works as follows. The adversary works as the server for KE interacting with any number of users as discussed above. Then any time two sessions accept, the attacker can start interacting with them in the way proscribed by $\mathcal{G}$. The same secret bit b is shared across all pairs of sessions that have accepted (and not used anywhere else). The

$\text{Game } \mathrm{G}^{\mathsf{oke}\text{-}\mathcal{G}}_{\Pi,\mathsf{KE},\mathcal{P},b}(\mathcal{A})$
For $u \in \{0,1\}^*$ do
  $st_u \leftarrow\!\$\ \mathsf{KE.PKg}(u)$
$b' \leftarrow\!\$\ \mathcal{A}^{\mathcal{K},\textsc{Exp},\mathrm{O}}$
For $\mathsf{lab} \in \mathcal{U}$ do
  If $\mathcal{G}.\mathbf{Pred}(\mathbf{T}[\mathsf{lab}])$
    $b' \leftarrow 0$
If $\mathsf{V}(\mathbf{T})$ then $b' \leftarrow 0$
Return $(b' = 1)$

$\textsc{Exp}(u,i)$
Require $sst_u^i.\mathsf{status} = \mathsf{accepted}$
$\mathsf{lab}^* \leftarrow (u,i)$
If $\mathsf{lab}^* \notin \mathcal{U}$
  $\mathsf{lab}^* \leftarrow (sst_u^i.\mathsf{pid}, sst_u^i.\mathsf{pid})$
$\mathbf{T}[\mathsf{lab}^*].\mathsf{add}(\textsc{Exp},(u,i),\vec{k}[(u,i)])$
Return $\vec{k}[(u,i)]$

$\mathrm{O}(x,u,i)$
Require $(u,i) \in \mathcal{U}$
$\mathsf{lab} \leftarrow (u,i)$
$\mathsf{lab}' \leftarrow (sst_u^i.\mathsf{pid}, sst_u^i.\mathsf{psess})$
$in \leftarrow \sigma_{\mathcal{G}}[\mathsf{lab}], \vec{k}[\mathsf{lab}], \vec{k}[\mathsf{lab}'], x$
$(\sigma_{\mathcal{G}}[\mathsf{lab}], \vec{k}[\mathsf{lab}], \vec{k}[\mathsf{lab}'], y)$
  $\leftarrow\!\$\ \mathcal{G}^{\Pi}(in)$
$\mathbf{T}[\mathsf{lab}].\mathsf{add}(\mathrm{O},x,y)$
Return y

$\textsc{Fing}(u,i,v,j)$
If $sst_u^i.\mathsf{fing} = sst_v^j.\mathsf{fing}$ then
  $sst_u^i.\mathsf{status} \leftarrow \mathsf{accepted}$; $sst_v^j.\mathsf{status} \leftarrow \mathsf{accepted}$
  $sst_u^i.\mathsf{psess} \leftarrow j$; $sst_v^j.\mathsf{psess} \leftarrow i$
  $\vec{k}[(u,i)] \leftarrow sst_u^i.\mathsf{keys}$; $\vec{k}[(v,j)] \leftarrow sst_v^j.\mathsf{keys}$
  If $sst_u^i.\mathsf{role} = \mathcal{I}$ then $\mathsf{lab}^* \leftarrow (u,i)$
  Else $\mathsf{lab}^* \leftarrow (v,j)$
  $\mathcal{U} \leftarrow \mathcal{U} \cup \{\mathsf{lab}^*\}$; $\sigma_{\mathcal{G}}[\mathsf{lab}^*] \leftarrow\!\$\ \mathcal{G}.\mathbf{Start}(b)$; $t \leftarrow \mathsf{I}(\mathbf{T},u,i,v,j)$
  For $(w,k) \in t$ do $\mathbf{T}[\mathsf{lab}^*].\mathsf{add}(\textsc{Exp}, sst_w^k.\mathsf{role}, \vec{k}[(w,k)])$
Else
  $sst_u^i.\mathsf{status} \leftarrow \mathsf{rejected}$; $sst_v^j.\mathsf{status} \leftarrow \mathsf{rejected}$
$\mathbf{T}.\mathsf{add}(\textsc{Fing},(u,i,v,j),\varepsilon)$; Return ε

Figure 3.3. Security game measuring the security of a messaging scheme Π when its keys are produced by OKE protocol KE. For compactness we define $\mathcal{K} = \textsc{Reg}, \textsc{Req}, \textsc{Fin}, \textsc{Fing}, \textsc{Corrupt}, \textsc{Reveal}$.

---

attacker's goal is then to guess this secret bit.

Explanation of security game.

Note that the security game is parameterized by parameter $\mathcal{P}$. The parameter specifies a tuple of two algorithms (I,V). The former is used to precisely capture the mapping of when corruption/reveals of state used by the key exchange correspond to exposures of the initial keys

of the messaging protocol. The latter is used to place restrictions on what sequences of queries are allowed from the attacker. The extent to which KE achieves things like forward security or post-compromise security depend on the choice of these algorithms. Additionally, the restrictions of V are used to prevent sequences of queries that do not make sense. We will discuss them in more detail momentarily while discussing the behavior of the game.

In the game, we first initialize the state st_u of all users $u \in \{0,1\}^*$. In pseudocode this is written as an infinite loop. The formal interpretation of this is that each st_u will be lazily sampled. For any user u, immediately before the first time st_u would be used we sample it. Then the adversary $\mathcal{A}$ is run with access to eight oracles. The first six of these correspond to interacting with the key exchange protocol in the manner discussed while the last two are how the adversary interacts with the underlying messaging protocol after users have generated keys.

Oracles REG, REQ, FIN, CORRUPT, and REVEAL are defined as specified in Fig. 3.2. A transcript $\mathbf{T}$ is used to keep track of all of the adversaries interaction with the key exchange protocol. The details of FING are a bit more complicated because its details capture the link between KE and Π. The oracle takes as input sessions (u,i) and (v,j). The fingerprints of these sessions are compared. If they match, then we set up any "instance" of $\mathrm{G}^{\mathcal{G}}_{\Pi,b}$ for the keys they generated to be used in. Otherwise these sessions reject (by updating their statuses to rejected).

Let's walk through the behavior of FING when the fingerprints match. First we do some bookkeeping updates of sst_u^i and sst_v^j by setting their statuses to accepted and setting them as each others partner. Next we need to prepare for their keys to be used by Π. The important variables used by $\mathrm{G}^{\mathcal{G}}_{\Pi,b}$ were a transcript $\mathbf{T}$, game state $\sigma_{\mathcal{G}}$, and keys $\vec{k}[\mathcal{I}]$ and $\vec{k}[\mathcal{R}]$. We create these now, separating them from other instances of the game by labelling them with the sessions that use them. For the keys this is straightforward because they can just be labelled by the session that created them. For the other variables we adopt the convention that they will be labelled by whichever user is $\mathcal{I}$ which is achieved via the label variable lab^* (which we store in a set $\mathcal{U}$ to track each game instance). The final interesting detail here is how we (possibly) add initial exposures to the transcript $\mathbf{T}[\mathsf{lab}^*]$. This is done via the algorithm I. It is told the global transcript

$\mathbf{T}$ and which sessions are now being linked. Based on this it decides if $\mathcal{A}$ has done corruptions or reveals breaching the security of these current sessions. It outputs this decision in a list t which we then iterate over to add the corresponding exposures to $\mathbf{T}[\mathsf{lab}^*]$. It is required that $\mathsf{I}(T,u,i,v,j) \subseteq \{(u,i),(v,j)\}$ always.

After sessions have been accepted they can then be interacted with via the oracles EXP and O which just mirror the corresponding oracles from $\mathrm{G}^{\mathcal{G}}_{\Pi,b}$. All instances of this game were initiated with the same secret bit b. The goal of $\mathcal{A}$ is then to guess this bit. Its guess is overwritten if it made a disallowed sequence of queries. Whether this occurred is captured by checking each $\mathbf{T}[\mathsf{lab}^*]$ with $\mathcal{G}$.**Pred** and the key exchange transcript with V.

We require as a minimum that V enforce the following restrictions.

1. A user must be registered before being interacted with in any manner.
2. Request and finalize cannot be performed by sessions that have already been created.
3. Sessions must be created and not have rejected before being used in a fingerprint comparison. A single session cannot perform fingerprint comparisons twice.
4. Sessions which compare fingerprints must have the correct values for pid. A session cannot compare fingerprints with itself.
5. A session can only be revealed if it has already been created and it has not accepted or rejected. (After an accepting fingerprint comparison the session state stored by the key exchange protocol is given away to the messaging protocol and no longer stored. After a rejecting fingerprint comparison the session state should be deleted.)

It is important that each of these can easily be verified given only the transcripts of queries made. We refer to the V which enforces these restrictions and nothing more as V^*. Note that placing addition restrictions would correspond to *weakening* the security definition. We define V^* more formally in Section 3.7.

Security in the face of compromise.

As mentioned earlier, the particular choice of I and V determines what level of security KE is providing against the exposure of its secrets. At an informal level, there are three types of this security that an OKE protocol might achieve. These are "normal" forward security, forward security with respect to REG, and post-compromise security with respect to REG. The first of these means that corrupting a user will not breach security of any of its sessions that have already been created. The second means that corrupting a user after a registration will not breach security of any sessions that were created before the registration. The third of these means that corrupting a user will not breach security of any sessions created after the next registration. A final related notion we will informally call "security against $\mathcal{R}$ compromise" requires that corrupting a user does not breach security of any of its sessions which have the role $\mathcal{R}$.

The I algorithm which capture all of these simultaneously could behave as follows. First, it checks the transcript to determine the role of (u,i) and (v,j) by seeing whether they were created by REQ or FIN query. Then it traverses the transcript in order, checking each CORRUPT and REVEAL. On a corruption of u the session (u,i) is added to t if: its role is not $\mathcal{R}$ (security against $\mathcal{R}$ compromise), the most recent $\text{REG}(u)$ query is same as the $\text{REG}(u)$ query preceding the creation of (u,i) (forward and post-compromise security with respect to REG), and the session had not already been created at the time of the corruption (normal forward security). On a reveal of the session (u,i) it is always added to t. The session (v,j) is treated analogously. We refer to the I which behaves in this manner as I^*. We define I^* more formally in Section 3.7.

Not achieving all of these notions of security would be formalized by loosening the restrictions of which corruptions add sessions to t. Alternatively if a particular KE is weak to certain types of attacks this could be formalized by using V to disallow sequences of queries corresponding to those attacks.

Fingerprint usage in practice.

In this security definition we require that *all* users check fingerprints and, moreover, that they do so *before* commencing in using the messaging protocol. This, of course, is unrealistic. Actual users quite often do not check fingerprints and the whole point of the key exchange being *offline* is so that communicating users do not have to both be available at the same time to complete a key exchange. How, then, can our security notion be said to imply anything about the real world usage of the key exchange and messaging protocol?

We argue that it actually still implies quite a bit which we explain via several examples varying in how trusted the service provider is. A key observation is that when the fingerprints of two sessions are equal, *it does not matter if the comparison is ever actually done*. A failure in the fingerprint comparison tells the users that something has gone wrong and they should stop use of this sessions, but a successful comparison just tells them to continue on using the protocol as they've already been doing.

As a first example, suppose that a user trusts the service provider to follow the actual protocol specification in its behavior. This would correspond to an "honest-but-curious" adversary which could be formalized to require that all messages sent by the adversary be generate by honest executions of the server's algorithms. In this case, fingerprints are always equal so the above observation applies.

A less trustworthy service provider might be willing to diverge from the protocol as long as they cannot be detected doing so by the users. Certainly service providers have economic incentives not to be caught attacking their users. Such undetectable adversaries could be formalized in the style of algorithm substitution attacks [11] by requiring that an honest user cannot distinguish between interacting with an honest behaving server and the attacker. Since fingerprint mismatches never occur during honest execution an attacker which causes them would be detectable. But then fingerprints are again always equal so the above observation applies.

Of course, the above assumptions break down when the attacker is thought of not as the actual service provider, but as some malicious party which has broken into their systems and

taken control of the servers. Then the incentive to be undetectable may not be as strong.[1] In this case security is still assured for all sessions that have matching fingerprints (whether or not the users actual do the comparison). Security is not assured otherwise, but no security *can* be provided for sessions that run with mismatched fingerprints. Note that a malicious server can trivially run a man-in-the-middle attack against sessions that do not compare fingerprints. [2]

Ideal model caveat.

There is a small issue in the above formalism when capturing a multi-user version of $\mathrm{G}^{\mathcal{G}}$ if the later was defined in an ideal model (e.g. the random oracle model [14]). In $\mathrm{G}^{\mathcal{G}}$ this can be captured by O emulating a random oracle for queries of the form $x = (\textsc{Ro}, z)$. However, then each messaging session in $\mathrm{G}^{\mathsf{oke}\text{-}\mathcal{G}}$ would have access to its own random oracle which is independent from all of the other random oracle. The standard way of capturing the random oracle model would instead have the same random oracle shared between all users. We omit the details of formalization, but this issue can be generically solved using standard domain separation technique so the shared random oracle "acts" like a separate random oracle for each session.

3.3.2 Stand-alone security of OKE

Our ultimate goal is to find OKE protocols and messaging protocols which are secure when used together in the sense just described. Towards this, one could directly analyze any particular pair of KE and Π with respect each desired security specification $\mathcal{G}$. However, considering a number of Π and $\mathcal{G}$ have already been proposed, this would be rather inefficient. Instead it is much preferable to follow the standard of modular analysis and define a notion of what it means for an OKE protocol to be secure in isolation. We will do so now, taking care to choose the definition to allow a composition result implying that security of a messaging

[1] Though they likely still have some incentive to be undetectable so the service provider doesn't notice they have been breached.

[2] On close observation, one might be worried by the fact that in our security game the adversary cannot cause these sessions with mismatched fingerprints to run the messaging protocol at all. Hypothetically, an attacker might somehow be able to exploit observing these mismatched session to attack other session. However, these attacks actually are captured in our security by considering an attacker who doesn't call FING for sessions with mismatched fingerprints but instead uses REVEAL and emulates the behavior of the messaging protocol with the revealed keys.

protocol is maintained when used with a secure OKE protocol. That way security of a combined KE and Π can be assured by isolated analysis of each component separately.

Simulation-based security.

One's first instinct may be to write a security definition inspired by the original Bellare-Rogaway security definition for key exchange [15]. This style of definition is used quite regularly in the analysis of key exchange protocols and a strong composition result has already been proven [23].

Unfortunately, a typical Bellare-Rogaway style indistinguishability definition would be too strong for our purposes. Such definitions require that the keys generated by a pair of users during a key exchange are indistinguishable from fresh random keys. However, the most efficienct protocols for our setting will leak ℓ which is assumed to not break the security of the underlying messaging protocol. Because ℓ is typically correlated with the keys established this means an attacker would be able to trivially distinguish between the generated keys and fresh random keys simply by checking whether they are consistent with ℓ. Moreover, it is unclear how one would write such a definition to capture the intuition that a compromise of a party during key exchange corresponds *only* to an exposure of initial state used during the key exchange protocol.

To resolve this we use a simulation style definition. In such a definition we consider two "worlds" that an attacker might be in: a "real world" where it actually interacting with the key exchange protocol and an "ideal" world where it is interacting with a simulator. The goal of the attacker is to distinguish between the two. Security requires that for every attacker there exists a simulator such that the attacker cannot distinguish between a real key exchange and a simulation of it by the simulator given only ℓ.

Security definition.

Our notion of security is captured by the game $\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},b}(\mathcal{A})$. This game is parameterized by parameter $\mathcal{P} = (\mathsf{I},\mathsf{V})$, a simulator S, and a bit b. The goal of $\mathcal{A}$ is to guess the bit b, so its advantage with respect to S is defined by $\mathsf{Adv}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S}}(\mathcal{A}) = \Pr[\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},1}(\mathcal{A})] - \Pr[\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},0}(\mathcal{A})]$.

Because we are working in a concrete security setting there is no precise definition of security. Informally, KE is secure if for all efficient $\mathcal{A}$ there exists an efficient S such that $\mathsf{Adv}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S}}(\mathcal{A})$ is small. We think of $b = 1$ as the real world and $b = 0$ as the ideal world and describe each of these separately.

Real world.

Fig. 3.4 shows pseudocode for a real-world interaction of an adversary with a key exchange protocol. Oracles $\textsc{Reg}_1$, $\textsc{Req}_1$, $\textsc{Fin}_1$, $\textsc{Corrupt}_1$, and $\textsc{Reveal}_1$ are omitted from the figure and instead defined to be equal to the corresponding oracles specified for $\mathrm{G}^{\mathsf{oke}\text{-}\mathcal{G}}$ in Fig. 3.2. First $\textsc{Init}_1$ is run to generate st_u for each u. Note this matches how game $\mathrm{G}^{\mathsf{oke}\text{-}\mathcal{G}}$ began; the infinite pseudocode loop should be interpreted in the same way. Then $\mathcal{A}$ is given access to six oracles with which to interact with KE. All of them are identical to the oracles previously introduced for the composed security game except for $\textsc{Fing}_1$. Because we are studying KE in isolation this oracle no longer needs to prepare variables for use interacting with a messaging protocol Π. Instead if the fingerprints of the specified sessions match, the keys they derived are simply returned to the attacker. Note that these keys are not added to the transcript so that it mirrors the transcript from the composed game. The crux of the power of the security definition come from restrictions placed on how S is allowed to emulate this output in the ideal world.

Following all this interaction the adversary outputs a guess of the secret bit. As in the composed game, if the adversary has made invalid oracle queries the algorithm V can cause this guess to be overwritten.

Ideal world.

The ideal world is defined by the oracles shown in Fig. 3.5. It is defined with respect to a simulator S which attempts to simulate the output of the oracles from the real world given access to $\mathcal{O} = \mathrm{I}\textsc{Reg}, \mathrm{I}\textsc{Req}, \mathrm{I}\textsc{Fin}, \mathrm{I}\textsc{Reveal}$ which are defined in Fig. 3.6. The behavior of this is relatively straightforward in oracles $\textsc{Reg}_0$, $\textsc{Req}_0$, $\textsc{Fin}_0$, $\textsc{Corrupt}_0$, and $\textsc{Reveal}_0$. In each the simulator is told what query was just made by $\mathcal{A}$ and is given its current state. Based on this

$\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},b}(\mathcal{A})$	$\underline{\text{FING}_1(u,i,v,j)}$

$\underline{\text{INIT}_b}$
$b' \leftarrow_\$ \mathcal{A}^{\text{REG}_b,\text{REQ}_b,\text{FIN}_b,\text{FING}_b,\text{CORRUPT}_b,\text{REVEAL}_b}$
If $\mathsf{V}(\mathbf{T})$ then $b' \leftarrow 0$
Return $(b'=1)$
 	
$\underline{\text{INIT}_1}$
For $u \in \{0,1\}^*$ do $st_u \leftarrow_\$ \mathsf{KE.PKg}(u)$
	If $sst^i_u.\mathsf{fing} = sst^j_v.\mathsf{fing}$ then
	$sst^i_u.\mathsf{status} \leftarrow \mathsf{accepted}$
	$sst^j_v.\mathsf{status} \leftarrow \mathsf{accepted}$
	$sst^i_u.\mathsf{psess} \leftarrow j$
	$sst^j_v.\mathsf{psess} \leftarrow i$
	$z \leftarrow (sst^i_u.\mathsf{keys}, sst^j_v.\mathsf{keys})$
	Else
	$sst^i_u.\mathsf{status} \leftarrow \mathsf{rejected}$
	$sst^j_v.\mathsf{status} \leftarrow \mathsf{rejected}$
	$z \leftarrow (\bot,\bot)$
	$\mathbf{T}.\mathsf{add}(\text{FING},(u,i,v,j),\varepsilon)$
	Return z

Figure 3.4. Real world in OKE security model. Some oracles are omitted because they are identical to previously specified oracles.

---

is specifies what the output of the oracle will be while potentially updating its state. The other oracle are more complicated and require understanding the role that $\mathcal{O}$ is playing.

The oracles $\mathcal{O}$ and IFING can be thought of as an idealized key exchange being run by a trusted party which only leaks the public components of the exchanged keys. “Sessions” in this idealized key exchange are identified by labels (lab) controlled by the simulator. The session state of session lab is stored in $mst[\mathsf{lab}]$ and the keys it derives are stored in $\vec{k}[\mathsf{lab}]$. The values in mst are largely administrative values used by the “Require” statements at the beginning of oracles which prevent sessions from being used in invalid ways. Each of the oracles can be thought of as an “ideal” mirror to the corresponding protocols of KE. These oracle are similar in spirit to (and were inspired by) the oracles INITS, INITP, and INITK used in the definition of symmetric key protocols given by Brzuska, et. al. [23].

An ideal key exchange between lab_1 and lab_2 would consist of the following operations: $\text{IREG}(\mathsf{lab}_1)$; $\text{IREQ}(\mathsf{lab}_1,\mathsf{lab}_2)$; $\text{IFIN}(\mathsf{lab}_1,\mathsf{lab}_2)$; $\text{IFING}(\mathsf{lab}_1,\mathsf{lab}_2)$. With IREG the session lab_1 creates its half of the keys corresponding to a key exchange. The public part of its keys leak. We can roughly think of this as corresponding to a user registering with the server in the real world. Then with IREQ the session lab_2 is partnered with lab_1. It creates its own half of the

$\underline{\text{CORRUPT}_0(u)}$
$(\sigma, st_u) \leftarrow\!\!\$\ \mathsf{S}^{\mathcal{O}}(\sigma, \text{CORRUPT}, u)$
$\mathbf{T}.\mathsf{add}(\text{CORRUPT}, u, st_u)$
Return st_u

$\underline{\text{REVEAL}_0(u, i)}$
$(\sigma, \vec{k}) \leftarrow\!\!\$\ \mathsf{S}^{\mathcal{O}}(\sigma, \text{REVEAL}, u, i)$
$\mathbf{T}.\mathsf{add}(\text{REVEAL}, (u, i), \vec{k})$
Return $\vec{k}$

$\underline{\text{INIT}_0}$
$\sigma \leftarrow\!\!\$\ \mathsf{S}(\text{INIT})$

$\underline{\text{REG}_0(u)}$
$(\sigma, m) \leftarrow\!\!\$\ \mathsf{S}^{\mathcal{O}}(\sigma, \text{REG}, u)$; $\mathbf{T}.\mathsf{add}(\text{REG}, u, m)$; Return m

$\underline{\text{REQ}_0(u, i, m, v)}$
$(\sigma, m', fp) \leftarrow\!\!\$\ \mathsf{S}^{\mathcal{O}}(\sigma, \text{REQ}, u, i, m, v)$
$\mathbf{T}.\mathsf{add}(\text{REQ}, (u, i, m, v), (m', fp))$; Return (m', fp)

$\underline{\text{FIN}_0(u, i, m, v)}$
$(\sigma, fp) \leftarrow\!\!\$\ \mathsf{S}^{\mathcal{O}}(\sigma, \text{FIN}, u, i, m, v)$
$\mathbf{T}.\mathsf{add}(\text{FIN}, (u, i, m, v), fp)$; Return fp

$\underline{\text{FING}_0(u, i, v, j)}$
$(\sigma, y, \mathsf{lab}_1, \mathsf{lab}_2) \leftarrow\!\!\$\ \mathsf{S}^{\mathcal{O}}(\sigma, \text{FING}, u, i, v, j)$; $t \leftarrow \mathsf{I}(\mathbf{T}, u, i, v, j)$
$t' \leftarrow []$
For $\mathsf{lab} \in \mathcal{X}$ do
  If $\mathsf{lab} = \mathsf{lab}_1$ then $t'.\mathsf{add}((u, i))$
  If $\mathsf{lab} = \mathsf{lab}_2$ then $t'.\mathsf{add}((v, j))$
If y and $t = t'$ then
  $(\vec{k}_{\mathcal{I}}, \vec{k}_{\mathcal{R}}) \leftarrow \text{IFING}(\mathsf{lab}_1, \mathsf{lab}_2)$; $z \leftarrow (\vec{k}_{\mathcal{I}}, \vec{k}_{\mathcal{R}})$
Else $z \leftarrow (\bot, \bot)$
$\mathbf{T}.\mathsf{add}(\text{FING}, (u, i, v, j), \varepsilon)$; Return z

Figure 3.5. Ideal world in OKE security model. For compactness we let $\mathcal{O} = \text{IREG}, \text{IREQ}, \text{IFIN}, \text{IREVEAL}$.

---

keys corresponding to a key exchange and is given lab_1's half. The public part of lab_2's keys leak. This corresponds to the requesting protocol being run by the friend of that user. With IFIN the session lab_1 completes its side of the key exchange by receiving the half of its keys that were generated by lab_1. This corresponds to the first user running the finalization protocol. Finally with IFING both entities accept their exchange. This corresponds to a correct fingerprint comparison being performed.

The goal of the simulator is to emulate key exchange messages to make it look like sessions exchanged keys which match those produced by this ideal key exchange. The way it is

bound to matching the ideal key exchange is by oracle FING_1. In this oracle, rather than directly choosing the values returned to the adversary the simulator is required to choose sessions from the ideal key exchange whose keys will be returned.

Recall that a real world adversary can choose to learn the secret state of a particular user or session via CORRUPT_0 or REVEAL_0. For the simulator to be able to simulate this properly it must be able to compromise the secrets of the ideal key exchange. This is done via the exposure oracle IREVEAL which gives back all of the keys of a specified label and uses the table $\mathcal{X}$ to keep track of the fact that this has occurred. The intention is that a "well-behaved" simulator will not have exposed the labels it chooses for IFING unless the adversary $\mathcal{A}$ (via a CORRUPT or REVEAL) has compromised state that would allow it to know the keys derived by the exchange. This restriction is ensured in FING_0. If the lab's chosen by S were not exposed identically to how they should have been according to I then IFING is prevented from running.

Shadow keys.

In the above discussion we didn't address the fact that two copies of all keys are being stored. We refer to the second copy (indicated with a prime symbol) as the "shadow" keys. The shadow keys are identical to the real keys, except they store $k'_{\mathcal{R}}$ instead of $k_{\mathcal{R}}$. This key is given back to the simulator each time IREQ is called. If IREVEAL is queried on a session then its shadow keys are returned. During IFING, shadow keys are returned if either session has revealed. Otherwise, shadow keys are never used.

Shadow keys were introduced specifically to allow security against the following attack, while still enabling our composition result. Suppose the attacker honestly behaves as the server during a registration for u and then a request query for v trying to connect to u. Assuming a protocol which does leak the public keys, the simulator will already be "committed" to a choice of lab_1 for (u,i) and lab_2 for (v,j) which match the public keys leaked by the messages it has sent already. Because no secret state has been compromised at this point, the simulator should not have exposed either of these labels. Thus it does not know anything about $k_{\mathcal{R}}$. But at this

$\underline{\text{IREG}(\mathsf{lab})}$
Require $mst[\mathsf{lab}].\mathsf{status} = \bot$
$(pk_{\mathcal{I}}, sk_{\mathcal{I}}) \leftarrow\$ \Pi.\mathsf{Kg}_{\mathcal{I}}$; $\vec{k}[\mathsf{lab}] \leftarrow ((pk_{\mathcal{I}}, \bot), sk_{\mathcal{I}}, \bot)$
$\vec{k}'[\mathsf{lab}] \leftarrow ((pk_{\mathcal{I}}, \bot), sk_{\mathcal{I}}, \bot)$; $mst[\mathsf{lab}].\mathsf{status} \leftarrow \mathsf{created}$
$mst[\mathsf{lab}].\mathsf{role} \leftarrow \mathcal{I}$; Return $pk_{\mathcal{I}}$

$\underline{\text{IREQ}(\mathsf{lab}_1, \mathsf{lab}_2)}$
Require $mst[\mathsf{lab}_1].\mathsf{status} = \mathsf{created}$
Require $mst[\mathsf{lab}_2].\mathsf{status} = \bot$
$((pk_{\mathcal{I}}, \cdot), \cdot, \cdot) \leftarrow \vec{k}[\mathsf{lab}_1]$; $(pk_{\mathcal{R}}, sk_{\mathcal{R}}) \leftarrow\$ \Pi.\mathsf{Kg}_{\mathcal{R}}$
$k_{\mathcal{R}} \leftarrow\$ \{0,1\}^{\Pi.\mathsf{k}}$; $k'_{\mathcal{R}} \leftarrow\$ \{0,1\}^{\Pi.\mathsf{k}}$; $\ell \leftarrow (pk_{\mathcal{I}}, pk_{\mathcal{R}})$
$\vec{k}[\mathsf{lab}_2] \leftarrow (\ell, sk_{\mathcal{R}}, k_{\mathcal{R}})$; $\vec{k}'[\mathsf{lab}_2] \leftarrow (\ell, sk_{\mathcal{R}}, k'_{\mathcal{R}})$
$mst[\mathsf{lab}_2].\mathsf{status} \leftarrow \mathsf{waiting}$; $mst[\mathsf{lab}_2].\mathsf{role} \leftarrow \mathcal{R}$
$mst[\mathsf{lab}_2].\mathsf{pid} \leftarrow \mathsf{lab}_1$; Return $(pk_{\mathcal{R}}, k'_{\mathcal{R}})$

$\underline{\text{IFIN}(\mathsf{lab}_1, \mathsf{lab}_2)}$
Require $mst[\mathsf{lab}_1].\mathsf{status} = \mathsf{created}$
Require $mst[\mathsf{lab}_2].\mathsf{status} \notin \{\bot, \mathsf{created}\}$
$((pk_{\mathcal{I}}, \cdot), sk_{\mathcal{I}}, \cdot) \leftarrow \vec{k}[\mathsf{lab}_1]$; $((\cdot, pk_{\mathcal{R}}), \cdot, k_{\mathcal{R}}) \leftarrow \vec{k}[\mathsf{lab}_2]$
$((\cdot, \cdot), \cdot, k'_{\mathcal{R}}) \leftarrow \vec{k}'[\mathsf{lab}_2]$; $\vec{k}[\mathsf{lab}_1] \leftarrow ((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk_{\mathcal{I}}, k_{\mathcal{R}})$
$\vec{k}'[\mathsf{lab}_1] \leftarrow ((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk_{\mathcal{I}}, k'_{\mathcal{R}})$
$mst[\mathsf{lab}_1].\mathsf{status} \leftarrow \mathsf{waiting}$; $mst[\mathsf{lab}_1].\mathsf{pid} \leftarrow \mathsf{lab}_2$; Return ε

$\underline{\text{IREVEAL}(\mathsf{lab})}$
Require $mst[\mathsf{lab}].\mathsf{status} \notin \{\bot, \mathsf{accepted}\}$
$\mathcal{X}.\mathsf{add}(\mathsf{lab})$; Return $\vec{k}'[\mathsf{lab}]$

$\underline{\text{IFING}(\mathsf{lab}_1, \mathsf{lab}_2)}$
Require $mst[\mathsf{lab}_1].\mathsf{status} = \mathsf{waiting}$
Require $mst[\mathsf{lab}_2].\mathsf{status} = \mathsf{waiting}$
Require $mst[\mathsf{lab}_1].\mathsf{pid} = \mathsf{lab}_2$
Require $mst[\mathsf{lab}_2].\mathsf{pid} = \mathsf{lab}_1$
$mst[\mathsf{lab}_1].\mathsf{status} \leftarrow \mathsf{accepted}$; $mst[\mathsf{lab}_2].\mathsf{status} \leftarrow \mathsf{accepted}$
If $\{\mathsf{lab}_1, \mathsf{lab}_2\} \cap \mathcal{X} \neq \emptyset$ then return $(\vec{k}'[\mathsf{lab}_1], \vec{k}'[\mathsf{lab}_2])$
Return $(\vec{k}[\mathsf{lab}_1], \vec{k}[\mathsf{lab}_2])$

Figure 3.6. Oracles of an ideal key exchange.

point the simulator (in REQ_0) has sent a message which u should be able to "decrypt" to receive $k_{\mathcal{R}}$. In particular, if the attacker now reveals (v, j) it will receive $k_{\mathcal{R}}$. Then if it corrupts u the simulator needs to be able to produce a st_u which will correctly "decrypt" the message to $k_{\mathcal{R}}$.

Thus (in the absence shadow keys) security against such an attack would likely require some form of non-committing encryption [3, 52]. Nielson [52] showed that non-interactive

non-committing encryption is not possible even in non-programmable random oracle model. It is possible in the programmable random oracle model, but we would prefer to avoid our security definition requiring idealized model because there is no reason to believe that idealized models are required for security in the combined security game which is our ultimate goal.

A similar issue arises in the composition result proven by Brzuska, et. al. [23]; however, they are able to use a very different method of resolving it. In particular, the protocol being composed with the key exchange is not thought of as providing *any* security in the face of state compromise. The Bellare-Rogaway style security definition is not required to provide any guarantees about generated keys whose values are exposed, but still suffices for composed security because in the combined security game the attacker is allowed to arbitrarily choose the keys that will be used by any session which has been exposed. We cannot use such a technique because our primary motivation is providing continued security even when keys are exposed and the existing security games we are composing with do not provide the attacker with the capability to choose keys [17, 56, 44, 35, 45, 2]. It may be worth considering incorporating such an ability into the security definitions of messaging protocol because this ability would have allowed us to use a key encapsulation method in our coming construction instead of being forced to use full public key encryption. In the simulation style key exchange notion with *strong adaptive corruptions* of Shoup [61] a similar issue is resolved by disallowing the analogous sequence of oracle queries by the attacker.

3.4 Generic Construction

In this section we present our construction which generically provides a secure OKE scheme for any splittable initialization algorithm. Because we assume nothing of the initialization algorithm we are fairly constrained in what we can do to exchange the public keys. Consequently our construction has some inefficiencies which are likely inherent when working generically. We leave improving on this for specific distributions of public keys to future work.

Our generic construction GKE is as follow. Let splittable initialization algorithm GKE.Init (specified by $\mathsf{GKE.Kg}_{\mathcal{I}}$, $\mathsf{GKE.Kg}_{\mathcal{R}}$, and $\{0,1\}^{\mathsf{GKE.k}}$) be fixed. Let PKE be a public key encryption scheme and $n \in \mathbb{N}$.[3] Let GKE.SKg initializes $st_{\mathcal{S}}$ as an empty table and $\mathsf{GKE.PKg}(u)$ initializes st_u as a tuple of four empty tables. The rest of GKE's algorithms are as follows.

```
GKE.UReg(st)                                    GKE.SReg(st_S, m, u)
For l = 1,...,n do                              st_S[u] ← (m, 0)
   (pk[l], sk[l]) ←$ GKE.Kg_I                   Return st_S
   (ek[l], dk[l]) ←$ PKE.Kg
Return ((pk, sk, ek, dk), (pk, ek))
```

For registration a user u runs $\mathsf{GKE.Kg}_{\mathcal{I}}$ and PKE.Kg a total of n times to create lists of the corresponding keys. The list of public keys (**pk** and **ek**) are sent to the server which stores them. On a request by v for u to the server simply sends the next (pk, ek) in the list to v. Then v runs $\mathsf{GKE.Kg}_{\mathcal{R}}$ to obtain its key pair and uses ek to encrypt the $k_{\mathcal{R}}$ it samples. Then it sends $pk_{\mathcal{R}}$ and the ciphertext to the server which stores them until u requests a finalization with v. The fingerprint it simply set to be the hash of the sequence of messages seen by the users during the exchange.

```
GKE.SReq_1(st_S, v, u)                          GKE.SFin(st_S, u, v)
((pk, ek), l) ← st_S[u]                         m ← st_S[u, v].pop()
If l > n then return (st_S, ⊥)                  Return (st_S, m)
st_S[u] ← ((pk, ek), l)
Return (st_S, (pk[l], ek[l], l + 1))            GKE.UFin(st, m)
                                                (pk_R, c, l) ← m
GKE.UReq(st, m)                                 (pk, sk, ek, dk) ← st
If m = ⊥ then                                   If pk[l] = ⊥ then return (st, ⊥, ⊥)
   Return (st, ⊥, ⊥, ⊥)                         k_R ← PKE.Dec(dk[l], c)
(pk, ek, l) ← m                                 k⃗ ← ((pk[l], pk_R), sk[l], k_R)
(pk_R, sk_R) ←$ GKE.Kg_R                        (pk, ek) ← (pk[l], ek[l])
k_R ←$ {0,1}^GKE.k                              fp ← H(pk || ek || pk_R || c || l)
k⃗ ← ((pk, pk_R), sk_R, k_R)                     (pk[l], sk[l], ek[l], dk[l])
c ←$ PKE.Enc(ek, k_R)                              ← (⊥, ⊥, ⊥, ⊥)
fp ← H(pk || ek || pk_R || c || l)              st ← (pk, sk, ek, dk)
Return (st, (pk_R, c, l), k⃗, fp)                Return (st, k⃗, fp)

GKE.SReq_2(st_S, m, v, u)
st_S[u, v].add(m)
Return st_S
```

[3] We write GKE[PKE, n] when we want to make these parameters explicit.

Security of GKE.

We claim that GKE meets our desired notion of security. Recall $(\mathsf{I}^*, \mathsf{V}^*)$ defined earlier in Section 3.3.1. We claim security with respect to $\mathcal{P} = (\mathsf{I}^*, \mathsf{V}^*)$. In particular the advantage, $\mathsf{Adv}^{\mathsf{oke}}_{\mathsf{GKE},\mathcal{P},\mathsf{S}}(\mathcal{A})$, of any efficient $\mathcal{A}$ is small when using the simulator specified in Appendix 3.6. For space reasons we will not provide a full proof of security, but instead sketch the high level intuition for why our simulation works.

The simulator uses the ideal key exchange to simulate messages in an intuitively straightforward way (care was required to precisely figure out the low level details of how it works). Whenever S needs to know a public key it obtains this by making the appropriate call to $\text{IR}\textsc{eg}$ or $\text{IR}\textsc{eq}$ except in $\text{R}\textsc{eq}$ if u was given an a message which does not correspond to anything it should have been given in a key exchange with v. In this case it simply generates $pk_{\mathcal{R}}$, $sk_{\mathcal{R}}$, and $k_{\mathcal{R}}$ locally. Encryption keys are generated on registration. The ciphertext produced during a request are simulated by encrypting the shadow key $k'_{\mathcal{R}}$.

Taken independently, it clear that the marginal distributions of the output of S for $\text{R}\textsc{eg}$, $\text{R}\textsc{eq}$, and $\text{F}\textsc{in}$ are correct. The crux of the proof lies in arguing about the keys returned by $\text{F}\textsc{ing}$, $\text{C}\textsc{orrupt}$, and $\text{R}\textsc{eveal}$. First note that collision resistance of H ensures that two fingerprints match only if their two sessions had the same view of an interaction, which is what S checks for fingerprints. Moreover the entropy of PKE ensure that no ek or c will ever be produce twice, so *at most* two sessions can match in this manner. From this is it easy to verify that all keys output by $\text{F}\textsc{ing}$ will have the correct distribution with the possible exception of $k_{\mathcal{R}}$ which requires further analysis.

For a given ciphertext c, suppose the attacker does not make a corruption queries to learn dk. In this case CPA security of PKE guarantees that $\mathcal{A}$ cannot learn anything about what key was encrypted in c. If $\mathcal{A}$ did make such a corruption query then it can decrypt the ciphertext, but then it just sees the same $k'_{\mathcal{R}}$ which will later be returned by $\text{F}\textsc{ing}$ (because S will query $\text{IR}\textsc{eveal}$ during that corruption). The attacker could also learn about $k'_{\mathcal{R}}$ via a $\text{R}\textsc{eveal}$ query, but in this case S will query $\text{IR}\textsc{eveal}$ so again $k'_{\mathcal{R}}$ would be the key output by $\text{F}\textsc{ing}$.

Lastly we need to verify the output of REVEAL and CORRUPT. For REVEAL note that $\sigma[u,i]$ will always already contain the correct $pk_{\mathcal{I}}$, $pk_{\mathcal{R}}$, and $k_{\mathcal{R}}$ for that session (the last of these being the shadow key). Thus the simulator only needs to learn sk from IREVEAL. Similar reasoning gives that in CORRUPT the simulator only needs to use IREVEAL to learn sk and only for those l for which $\mathbf{pk}[l]$ has not already been erased.

This constitutes the main ideas of the proof. Formalizing it is a matter of careful bookkeeping to capture these arguments precisely.

3.5 Composition

Our motivation in designing our stand alone security definition for an OKE was to carefully define it so that it was efficiently achievable yet strong enough to imply secure composition with any messaging protocol. We address the latter point in this section.

Theorem 13 (Informal)**.** *Assume $\mathcal{P}$ is "well behaved." Suppose* KE *is secure with respect to* $\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P}}$ *and* Π *is secure with respect to* $\mathrm{G}^{\mathcal{G}}_{\Pi}$*. Then they are secure with respect to* $\mathrm{G}^{\mathsf{oke}\text{-}\mathcal{G}}_{\Pi,\mathsf{KE},\mathcal{P}}$.

What it means for means for $\mathcal{P}$ to be "well behaved" is specified in the formal theorem statement in Section 15 based on definitions in Section 3.7.

We dedicate the rest of this section to sketching how this result is proven. First we introduce a multi-user variant of our generic state exposure game which can be proven to be implied by the single user variant.

Multi-user messaging security.

The multi-user state exposure game $\mathrm{G}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi,b}$ is defined in Fig. 3.7. The advantage of adversary $\mathcal{A}$ is defined by $\mathsf{Adv}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi}(\mathcal{A}) = \Pr[\mathrm{G}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi,1}(\mathcal{A})] - \Pr[\mathrm{G}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi,0}(\mathcal{A})]$.

This game can be understood as having taken the game $\mathrm{G}^{\mathsf{oke}\text{-}\mathcal{G}}_{\Pi,\mathsf{KE},\mathcal{P}}$ and changed the oracles for interacting with KE into oracles for the "ideal key exchange" of $\mathrm{G}^{\mathsf{oke}}$. In particular, IREG, IREQ, IFIN, and IREVEAL are defined to be equal to those oracles as specified in Fig. 3.6. Oracle IFING has been modified.

Game $\mathrm{G}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi,b}(\mathcal{A})$	$\mathrm{O}(x,\mathsf{lab})$

$b' \leftarrow_{\$} \mathcal{A}^{\mathcal{O},\mathrm{IFING},\mathrm{O},\mathrm{EXP}}$
For $\mathsf{lab} \in \mathcal{U}$ do
  If $\mathcal{G}.\mathbf{Pred}(\mathbf{T}[\mathsf{lab}])$
    $b' \leftarrow 0$
Return $(b' = 1)$
 	
$\mathrm{EXP}(\mathsf{lab})$
Require $mst[\mathsf{lab}].\mathsf{status}$
  $= \mathsf{accepted}$
$\mathsf{lab}^* \leftarrow \mathsf{lab}$
If $\mathsf{lab}^* \notin \mathcal{U}$
  $\mathsf{lab}^* \leftarrow mst[\mathsf{lab}].\mathsf{pid}$
$r \leftarrow mst[\mathsf{lab}].\mathsf{role}$
$\mathbf{T}[\mathsf{lab}^*].\mathsf{add}(\mathrm{EXP}, r, \vec{k}[\mathsf{lab}])$
Return $\vec{k}[\mathsf{lab}]$
	Require $\mathsf{lab} \in \mathcal{U}$
	$\mathsf{lab}' \leftarrow mst[\mathsf{lab}].\mathsf{pid}$
	$in \leftarrow \sigma_{\mathcal{G}}[\mathsf{lab}], \vec{k}[\mathsf{lab}], \vec{k}[\mathsf{lab}'], x$
	$(\sigma_{\mathcal{G}}[\mathsf{lab}], \vec{k}[\mathsf{lab}], \vec{k}[\mathsf{lab}'], y)$
	  $\leftarrow_{\$} \mathcal{G}^{\Pi}(in)$
	$\mathbf{T}[\mathsf{lab}].\mathsf{add}(\mathrm{O}, x, y)$
	Return y

$\mathrm{IFING}(\mathsf{lab}_1, \mathsf{lab}_2)$
Require $mst[\mathsf{lab}_1].\mathsf{status} = \mathsf{waiting}$
Require $mst[\mathsf{lab}_2].\mathsf{status} = \mathsf{waiting}$
Require $mst[\mathsf{lab}_1].\mathsf{pid} = \mathsf{lab}_2$
Require $mst[\mathsf{lab}_2].\mathsf{pid} = \mathsf{lab}_1$
$mst[\mathsf{lab}_1].\mathsf{status} \leftarrow \mathsf{accepted}$; $mst[\mathsf{lab}_2].\mathsf{status} \leftarrow \mathsf{accepted}$
If $mst[\mathsf{lab}_1].\mathsf{role} = \mathcal{I}$ then $\mathsf{lab}^* \leftarrow \mathsf{lab}_1$
Else $\mathsf{lab}^* \leftarrow \mathsf{lab}_2$
$\mathcal{U} \leftarrow \mathcal{U} \cup \{\mathsf{lab}^*\}$; $\sigma_{\mathcal{G}}[\mathsf{lab}^*] \leftarrow_{\$} \mathcal{G}.\mathbf{Start}(b)$
If $\{\mathsf{lab}_1, \mathsf{lab}_2\} \cap \mathcal{X} \neq \emptyset$ then
  $(\vec{k}[\mathsf{lab}_1], \vec{k}[\mathsf{lab}_2]) \leftarrow (\vec{k}'[\mathsf{lab}_1], \vec{k}'[\mathsf{lab}_2])$
For $\mathsf{lab} \in \mathcal{X} \cap \{\mathsf{lab}_1, \mathsf{lab}_2\}$ do
  $\mathbf{T}[\mathsf{lab}^*].\mathsf{add}(\mathrm{EXP}, mst[\mathsf{lab}].\mathsf{role}, \vec{k}[\mathsf{lab}])$
Return ε

Figure 3.7. State exposure game with ideal key exchange. For compactness we let $\mathcal{O} = \mathrm{IREG}, \mathrm{IREQ}, \mathrm{IFIN}, \mathrm{IREVEAL}$.

The following lemma is a key step in our composition proof.

Lemma 14 (Informal)**.** *Suppose* Π *is secure with respect to* $\mathrm{G}^{\mathcal{G}}_{\Pi}$. *Then* Π *is secure with respect to* $\mathrm{G}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi}$.

The proof of this lemma uses a combination of standard hybrid and index-guessing proofs. The hybrid argument is used to create an adversary $\mathcal{B}$ which makes at most one FING query and achieves advantage $\mathsf{Adv}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi}(\mathcal{B}) \geq 1/q_{\mathrm{FING}} \cdot \mathsf{Adv}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi}(\mathcal{A})$ where q_{IFING} is the number

of IFING oracle queries made by $\mathcal{A}$. It runs $\mathcal{A}$, forwarding on all oracle queries it makes, with the following exceptions. At the beginning $\mathcal{B}$ picks an index $i \in [q_{\text{FING}}]$ at random. For all FING queries before the i-th $\mathcal{B}$ instead exposes the specified sessions and emulates queries to EXP and O with these sessions using secret bit 0. For all FING queries after the i-th it behaves analogously except with secret bit 1. Only for the i-th query does it actually forward on the fingerprint. It outputs whatever bit $\mathcal{A}$ does.

Then we use an index-guessing argument to reduce from $\mathcal{B}$ to a single-user adversary $\mathcal{C}$. This adversary just guesses ahead of time which two sessions $\mathcal{B}$ will call FING for and which of the two (if either) will be revealed first. In simulates the oracles for these sessions with its own oracles and information. If it guessed incorrectly it outputs $b' = 0$, otherwise it outputs whatever $\mathcal{B}$ does. Standard analysis gives $\mathsf{Adv}_{\Pi}^{\mathcal{G}}(\mathcal{C}) \geq 1/(3q_{\text{IREG}}q_{\text{IREQ}}) \cdot \mathsf{Adv}_{\Pi}^{\mathsf{mu}\text{-}\mathcal{G}}(\mathcal{B})$.

Rest of the proof.

The rest of the proof of Theorem 13 is to show that $\mathrm{G}_{\mathsf{KE},\mathcal{P}}^{\mathsf{oke}}$ security can be used to reduce $\mathrm{G}_{\Pi,\mathsf{KE},\mathcal{P}}^{\mathsf{oke}\text{-}\mathcal{G}}$ security to $\mathrm{G}_{\Pi}^{\mathsf{mu}\text{-}\mathcal{G}}$ security. Let $\mathcal{A}_{\mathsf{KE},\Pi}$ be an adversary against the combined security game. Then we construct an adversary $\mathcal{A}_{\mathsf{KE}}$ as follows. First it samples a secret bit b_{Π}. Then it forward on all key exchange queries to its own oracles. When a successful fingerprint comparison is done, $\mathcal{A}_{\mathsf{KE}}$ then uses the keys it is given back to simulate EXP and O for $\mathcal{A}_{\mathsf{KE},\Pi}$ using b_{Π}. Finally it outputs 1 if $\mathcal{A}_{\mathsf{KE},\Pi}$ correctly guesses b_{Π} and 0 otherwise.

We let S be a simulator for this new adversary which we use to build a multi-user adversary $\mathcal{A}_{\Pi}$. It runs $\mathcal{A}_{\mathsf{KE},\Pi}$ and simulates all of its key exchange queries using S while forwarding on all of S's queries to the ideal key exchange. Queries to EXP and O are forwarded on, replacing (u,i) identifiers with the labels chosen by S. Then the difference between $\mathcal{A}_{\Pi}$'s advantage and $\mathcal{A}_{\mathsf{KE},\Pi}$'s is exactly the advantage of $\mathcal{A}_{\mathsf{KE}}$ which must be small from the assumed security of KE. Applying Lemma 14 finishes the proof.

3.6 Simulator Pseudocode

```
S^O(σ, Reg, u)
(·,·,·,t) ← σ[u] ; t ← t+1
For l = 1,...,n do
  pk[l] ←$ IReg((u,t,l))
  (ek[l], dk[l]) ←$ PKE.Kg
σ[u] ← (pk, ek, dk, t)
Return (σ, (pk, ek))

S^O(σ, Req, u, i, m, v)
If m = ⊥ then return (σ, ⊥, ⊥)
(pk, ek, dk, t) ← σ[v]
(pk, ek, l) ← m
If (pk, ek) = (pk[l], ek[l])
  (pk_R, k'_R)
    ← IReq((v,t,l), (u,i))
  σ[u,i].lab ← (u,i)
  σ[u,i].pid ← (v,t,l)
  sk_R ← ⊥
Else
  (pk_R, sk_R) ←$ KE.Kg_R
  k'_R ←$ {0,1}^KE.k
k⃗ ← ((pk, pk_R), sk_R, k'_R)
c ←$ PKE.Enc(ek, k'_R)
z ← pk ‖ ek ‖ pk_R ‖ c ‖ l
fp ← H(z)
σ[u,i].keys ← k⃗
σ[u,i].fing ← z
σ[u, pk, ek, pk_R, c, l] ← i
Return (σ, (pk_R, c, l), fp)
```

```
S^O(σ, Fin, u, i, m, v)
(pk, c, l) ← m
(pk, ek, dk, t) ← σ[u]
If pk[l] = ⊥ then return (σ, ⊥)
j ← σ[v, pk[l], ek[l], pk, c, l]
If σ[v,j].lab ≠ ⊥
  IFin((u,t,l), (v,j))
  σ[u,i].lab ← (u,t,l)
  σ[u,i].pid ← (v,j)
  sk_I ← ⊥
Else
  ((·,·), sk_I, ·) ← IReveal((u,t,l))
k_R ← PKE.Dec(dk[l], c)
k⃗ ← ((pk[l], pk), sk_I, k_R)
z ← pk[l] ‖ ek[l] ‖ pk ‖ c ‖ l
fp ← H(z)
(pk[l], ek[l], dk[l]) ← (⊥, ⊥, ⊥)
σ[u] ← (pk, ek, dk, t)
σ[u,i].keys ← k⃗
σ[u,i].fing ← z
Return (σ, fp)
```

$\underline{\mathsf{S}^{\mathcal{O}}(\sigma, \textsc{Fing}, u, i, v, j)}$
If $\sigma[u,i].\mathsf{fing} = \sigma[v,j].\mathsf{fing}$
  $y \leftarrow \mathsf{true}$; $\mathsf{lab}_1 \leftarrow \sigma[u,i].\mathsf{lab}$; $\mathsf{lab}_2 \leftarrow \sigma[v,j].\mathsf{lab}$
  If $\bot \in \{\mathsf{lab}_1, \mathsf{lab}_2\}$ then $y \leftarrow \mathsf{false}$
  If $\sigma[u,i].\mathsf{pid} \neq \mathsf{lab}_2$ or $\sigma[v,j].\mathsf{pid} \neq \mathsf{lab}_1$ then $y \leftarrow \mathsf{false}$
Else $(y, \mathsf{lab}_1, \mathsf{lab}_2) \leftarrow (\mathsf{false}, \bot, \bot)$
Return $(\sigma, y, \mathsf{lab}_1, \mathsf{lab}_2)$

$\underline{\mathsf{S}^{\mathcal{O}}(\sigma, \textsc{Corrupt}, u)}$
$(\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t) \leftarrow \sigma[u]$
For l s.t. $\mathbf{pk}[l] \neq \bot$ do $((\cdot,\cdot), \mathbf{sk}[l], \cdot) \leftarrow \textsc{IReveal}((u,t,l))$
Return $(\sigma, (\mathbf{pk}, \mathbf{sk}, \mathbf{ek}, \mathbf{dk}))$

$\underline{\mathsf{S}^{\mathcal{O}}(\sigma, \textsc{Reveal}, u, i)}$
$((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk, k_{\mathcal{R}}) \leftarrow \sigma[u,i].\mathsf{keys}$
If $\sigma[u,i].\mathsf{lab} \neq \bot$ then
  $((\cdot,\cdot), sk, \cdot) \leftarrow \textsc{IReveal}(\sigma[u,i].\mathsf{lab})$
  $\sigma[u,i].\mathsf{keys} \leftarrow ((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk, k_{\mathcal{R}})$
Return $(\sigma, \sigma[u,i].\mathsf{keys})$

3.7 Formalization of Parameters

In this section we more formally define the parameters V^* and I^* with respect to which we claim security.

Minimum V.

In Section 3.3.1 we defined V^* as enforcing the following minimal restrictions.

1. A user must be registered before being interacted with in any manner.

2. Request and finalize cannot be performed by sessions that have already been created.

3. Sessions must be created and not have rejected before being used in a fingerprint comparison. A single session cannot perform fingerprint comparisons twice.

4. Sessions which compare fingerprints must have the correct values for pid. A session cannot compare fingerprints with itself.

5. A session can only be revealed if it has already been created and it has not accepted or rejected. (After an accepting fingerprint comparison the session state stored by the key exchange protocol is given away to the messaging protocol and no longer stored. After a rejecting fingerprint comparison the session state should be deleted.)

We give the formal definition of V^* by providing concrete pseudocode for computing it in Fig. 3.8. To simplify notation we temporarily changed our convention of what "Require bool" means. In particular in V^* we use it as shorthand for the pseudocode "If not bool then $\mathsf{reject} \leftarrow \mathsf{true}$". Three useful properties of V^* are that it is monotonic and efficiently computable in an online manner.

We say a V is monotonic if the following for all $\mathbf{T}$ and $\mathbf{T}'$. If $\mathbf{T}$ is a prefix of $\mathbf{T}'$ and $\mathsf{V}(\mathbf{T}) = \mathsf{true}$, then $\mathsf{V}(\mathbf{T}') = \mathsf{true}$. Additionally, $\mathsf{V}(\mathbf{T}) = \mathsf{false}$ if $\mathbf{T}$ is the empty transcript.

We say a V is efficiently computable in an online manner if V is efficiently computable and $\mathsf{V}(\mathbf{T})$ does not depend on the output value of the last element of $\mathbf{T}$. More formally, there exists an efficient $\mathsf{V}(\cdot,\cdot,\cdot)$ such that $\mathsf{V}(\mathbf{T}, \mathrm{O}, x)$ returns true if and only if computing $\mathbf{T}.\mathsf{add}(\mathrm{O}, x, y)$; $\mathsf{V}(\mathbf{T})$ would return true for all y.

These properties allow us to restrict attention to adversaries that never produce transcripts for which $\mathsf{V}^*(\mathbf{T}) = \mathsf{true}$ because an adversary can check beforehand and refrain from making any queries which would cause this. In Theorem 15 we show composability only for V with these properties.

Maximal I^*.

In Section 3.3.1 we sketched the following algorithm for I^*.

- First, it checks the transcript to determine the role of (u,i) and (v,j) by seeing whether they were created by REQ or FIN query.

- Then it traverses the transcript in order while checking each CORRUPT and REVEAL.

- On a corruption of u the session (u,i) is added to t if its role is not $\mathcal{R}$, the most recent

```
V*(T)
reject ← false
For (Proc, x, y) ∈ T
   Proc(x, y)
Return reject

Reg(u, m)
reg_u ← true

Fin((u, i, m, v), fp)
Require reg_u
Require sst^i_u.status = ⊥
sst^i_u.fing ← fp
sst^i_u.pid ← v
If fp = ⊥ then
   sst^i_u.status ← rejected
Else
   sst^i_u.status ← waiting

Req((u, i, m, v), (m', fp))
Require reg_u
Require sst^i_u.status = ⊥
sst^i_u.fing ← fp
sst^i_u.pid ← v
If fp = ⊥ then
   sst^i_u.status ← rejected
Else
   sst^i_u.status ← waiting

Corrupt(u, st_u)
Require reg_u

Reveal((u, i), sst^i_u.keys)
Require reg_u
Require sst^i_u.status = waiting

Fing((u, i, v, j), ε)
Require reg_u
Require sst^i_u.status = waiting
Require sst^j_v.status = waiting
Require sst^i_u.pid = v
Require sst^j_v.pid = u
Require (u, i) ≠ (v, j)
If sst^i_u.fing = sst^j_v.fing then
   sst^i_u.status ← accepted ; sst^j_v.status ← accepted
Else
   sst^i_u.status ← rejected ; sst^j_v.status ← rejected
```

Figure 3.8. Pseudocode formalizing the minimal V.

$\text{Reg}(u)$ query is same as the $\text{Reg}(u)$ query preceding the creation of (u,i), and the session had not already been created at the time of the corruption.

- On a reveal of the session (u,i) it is always added to t.
- The session (v,j) is treated analogously during this traversal.

In Fig. 3.9 we provide precise pseudocode for I^*. It traverses the transcript three times. In the first traversal it figures out the role of both sessions as well as the time of creation. In the second

```
I*(T, u, i, v, j)
For l = 1, ..., |T| do
  (Proc, x, y) ← T[l]
  If Proc = Req and x = (u, i, ·, ·)
    sst_u^i.role ← R ; sst_u^i.time ← l
  If Proc = Req and x = (v, j, ·, ·)
    sst_v^j.role ← R ; sst_v^j.time ← l
  If Proc = Fin and x = (u, i, ·, ·)
    sst_u^i.role ← I ; sst_u^i.time ← l
  If Proc = Fin and x = (v, j, ·, ·)
    sst_v^j.role ← I ; sst_v^j.time ← l
For l = 1, ..., |T| do
  (Proc, x, y) ← T[l]
  If Proc = Reg and x = u and l < sst_u^i.time
    sst_u^i.prev ← l
  If Proc = Reg and x = v and l < sst_v^j.time
    sst_v^j.prev ← l
For l = 1, ..., |T| do
  (Proc, x, y) ← T[l]
  If Proc = Corrupt and x = u
    If sst_u^i.role ≠ R and sst_u^i.prev < l < sst_u^i.time
      t.add(u, i)
  If Proc = Corrupt and x = v
    If sst_v^j.role ≠ R and sst_v^j.prev < l < sst_v^j.time
      t.add(v, j)
  If Proc = Reveal and x = (u, i)
    t.add(u, i)
  If Proc = Reveal and x = (v, j)
    t.add(v, j)
Return t
```

Figure 3.9. Pseudocode formalizing the maximal I.

traversal it determines the time of the preceding REG for each session. Then in the third session it uses the stated logic to determine when to add sessions to t.

3.8 Composition proof

In this section we formalize and prove Theorem 13. This theorem shows that if a messaging scheme Π and OKE protocol KE are individually secure, then they are secure when

used in combination. The following theorem formalizes this intuition.

Theorem 15. *Let* KE *be an OKE protocol and* Π *be a messaging scheme. Let* $\mathcal{G}$ *be fixed. Let* $\mathcal{P} = (\mathsf{I}, \mathsf{V})$ *for some* V *which is monotonic and efficiently computable in an online manner. Let* $\mathcal{A}_{\Pi,\mathsf{KE}}$ *be an adversary making at most q oracle queries and* S *be a clean simulator. Then we can construct adversaries* $\mathcal{A}_{\mathsf{KE}}$ *and* $\mathcal{A}_{\Pi}$ *for which the following inequality holds.*

$$\mathsf{Adv}^{\mathsf{oke}\text{-}\mathcal{G}}_{\Pi,\mathsf{KE},\mathcal{P}}(\mathcal{A}_{\Pi,\mathsf{KE}}) \leq \mathsf{Adv}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S}}(\mathcal{A}_{\mathsf{KE}}) + 3q^3\mathsf{Adv}^{\mathcal{G}}_{\Pi}(\mathcal{A}_{\Pi})$$

Adversaries $\mathcal{A}_{\mathsf{KE}}$ *and* $\mathcal{A}_{\Pi}$ *are comparably efficient to* $\mathcal{A}_{\Pi,\mathsf{KE}}$ *and* S.

We say a simulator is clean if the number of $\text{IR}\textsc{eg}$ (resp. $\text{IR}\textsc{eq}$) queries it makes never exceeds the number of times it is run with input $\text{R}\textsc{eg}$ (resp. $\text{R}\textsc{eq}$). For non-clean simulators the same proof works by replacing $3q^3$ in the above bound with $3q_{\text{F}\textsc{ing}} \cdot q_{\text{IR}\textsc{eg}} \cdot q_{\text{IR}\textsc{eq}}$ where $q_{\text{F}\textsc{ing}}$ is an upper bound on the number of queries $\mathcal{A}_{\Pi,\mathsf{KE}}$ makes to $\text{F}\textsc{ing}$ and $q_{\text{IR}\textsc{eg}}$ (resp. $q_{\text{IR}\textsc{eq}}$) is an upper bound on the number of oracle queries that S makes to $\text{IR}\textsc{eg}$ (resp. $\text{IR}\textsc{eq}$).

As mentioned in Section 3.5, it is useful to first provide a lemma showing that security with respect to our multi-user state exposure game is implied by security with respect to the single user variant. The following lemma implies this and is a formalization of the informally stated Lemma 14.

Lemma 16. *Let* Π *be a messaging scheme. Fix* $\mathcal{G}$ *and let* $\mathcal{A}$ *be an adversary making at most* q_{X} *oracle queries to each of its oracles* X. *Then we can construct* $\mathcal{C}$ *such that*

$$\mathsf{Adv}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi}(\mathcal{A}) \leq 3q_{\text{IF}\textsc{ing}} \cdot q_{\text{IR}\textsc{eg}} \cdot q_{\text{IR}\textsc{eq}}\mathsf{Adv}^{\mathcal{G}}_{\Pi}(\mathcal{C}).$$

Adversary $\mathcal{C}$ *is comparably efficient to* $\mathcal{A}$.

Proof. To start the proof we reduce to the case that the adversary only ever makes one $\text{F}\textsc{ing}$ query via a standard hybrid argument. From there we provide our reduction to $\text{G}^{\mathcal{G}}$. This latter

```
Adversary B^{O,IFING,O,EXP}
q <-$ [q_IFING] ; ctr <- -1
b' <-$ A^{OS,OS,EXPS}
For lab ∈ U do
   If G.Pred(T[lab])
      b' <- 0
Return (b' = 1)

EXPS(lab)
lab* <- lab
If lab* ∉ U
   lab* <- mst[lab].pid
If ctr[lab*] = q
   Return EXP(lab)
r <- mst[lab].role
T[lab*].add(EXP, r, k⃗[lab])
Return k⃗[lab]

OS(x, lab)
If ctr[lab] = q
   Return O(x, lab)
lab' <- mst[lab].pid
in <- σ_G[lab], k⃗[lab], k⃗[lab'], x
(σ_G[lab], k⃗[lab], k⃗[lab'], y) <-$ G^Π(in)
T[lab].add(O, x, y)
Return y

IFINGS(lab_1, lab_2)
If mst[lab_1].role = I then lab* <- lab_1 else lab* <- lab_2
ctr <- ctr + 1 ; ctr[lab*] <- ctr
If i = q then IFING(lab_1, lab_2) ; Return ε
Else if i > q then σ_G[lab*] <-$ G.Start(0)
Else σ_G[lab*] <-$ G.Start(1)
U <- U ∪ {lab*}
k⃗'[lab_1] <- IREVEAL(lab_1) ; k⃗'[lab_2] <- IREVEAL(lab_2)
If {lab_1, lab_2} ∩ X ≠ ∅ then
   (k⃗[lab_1], k⃗[lab_2]) <- (k⃗'[lab_1], k⃗'[lab_2])
Else
   k <-$ {0,1}^{Π.k}
   (ℓ, sk, k') <- k⃗'[lab_1] ; k⃗[lab_1] <- (ℓ, sk, k)
   (ℓ, sk, k') <- k⃗'[lab_2] ; k⃗[lab_2] <- (ℓ, sk, k)
For lab ∈ X ∩ {lab_1, lab_2} do
   T[lab*].add(EXP, mst[lab].role, k⃗[lab])
Return ε
```

Figure 3.10. Adversary for proof of Lemma 16 making at most one query to IFING. Oracles $\mathcal{O}$S are described in the text.

---

reduction requires the adversary to guess beforehand which labels will be used in the FING query and whether either of their secrets will be revealed.

We assume without loss of generality that $\mathcal{A}$ never makes a query which triggers any of the "Require" statements. Our first adversary $\mathcal{B}$ is shown in Fig. 3.10. For the oracles of the ideal

key exchange ($\mathcal{O}$S) other than IFING we assume that $\mathcal{A}_1$ simply forwards the queries on to its own ideal key exchange oracles. While doing so it keeps track of the values of all values stored in mst and $\mathcal{X}$ (which is clearly straightforward to do). The other oracles are simulated as shown in the figure by IFINGS, EXPS, and OS.

Adversary $\mathcal{B}$ starts by sampling q at random from $[q_{\text{FING}}]$. The q-th query to IFINGS will be forwarded on to $\mathcal{B}$'s own oracles while all other queries will be simulated locally by $\mathcal{B}$. It uses the counter ctr to keep track of this. When $ctr[\mathsf{lab}^*] > q$ the state $\sigma_{\mathcal{G}}[\mathsf{lab}^*]$ it initialized with $\mathcal{G}.\mathbf{Start}(0)$ while $\mathcal{G}.\mathbf{Start}(1)$ is used if $ctr[\mathsf{lab}^*] < q$.

Let $\Pr[\mathcal{B}_x^b]$ denote the probability that $\mathrm{G}^{\mathsf{mu}\text{-}\mathcal{G}}(\mathcal{B})$ returns true conditioned on $\mathcal{B}$ having sampled $q = x$ and the underlying bit of $\mathrm{G}^{\mathsf{mu}\text{-}\mathcal{G}}$ being b. Thus $\mathsf{Adv}_{\Pi}^{\mathsf{mu}\text{-}\mathcal{G}}(\mathcal{B}) = \mathbf{E}_x(\Pr[\mathcal{B}_x^1]) - \mathbf{E}_x(\Pr[\mathcal{B}_x^0])$. We make the following three claims.

1. $\Pr[\mathcal{B}^1_{q_{\text{IFING}}-1}] = \Pr[\mathrm{G}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi,1}(\mathcal{A})]$
2. $\Pr[\mathcal{B}^0_0] = \Pr[\mathrm{G}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi,0}(\mathcal{A})]$
3. $\Pr[\mathcal{B}^1_x] = \Pr[\mathcal{B}^0_{x+1}]$ for all x

For claim (1) note that when $q = q_{\text{IFING}} - 1$ it will never hold that $ctr[\mathsf{lab}^*] > q$. This and $b = 1$ implies that $\mathcal{G}.\mathbf{Start}(1)$ was used to initiate all of the game states $\mathcal{A}$ interacts with as in $\mathrm{G}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi,1}$. For claim (2) note that when $q = 0$ it will never hold that $ctr[\mathsf{lab}^*] < q$. This and $b = 0$ implies that $\mathcal{G}.\mathbf{Start}(0)$ was used to initiate all of the game states $\mathcal{A}$ interacts with as in $\mathrm{G}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi,0}$. For claim (3) note that in both of these cases, $\mathcal{G}.\mathbf{Start}(0)$ is being used when $ctr[\mathsf{lab}^*] \leq x$ and $\mathcal{G}.\mathbf{Start}(1)$ is being used otherwise.

The claims give use the following.

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi}(\mathcal{A}) &= \Pr[\mathcal{B}^1_{q_{\text{IFING}}-1}] - \Pr[\mathcal{B}^0_0] \\
&= \Pr[\mathcal{B}^1_{q_{\text{IFING}}-1}] - \Pr[\mathcal{B}^0_0] + \sum_{x=0}^{q_{\text{IFING}}-2} (\Pr[\mathcal{B}^1_x] - \Pr[\mathcal{B}^0_{x+1}]) \\
&= \sum_{x=0}^{q_{\text{IFING}}-1} (\Pr[\mathcal{B}^1_x] - \Pr[\mathcal{B}^0_x]) = q_{\text{IFING}} \cdot \mathsf{Adv}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi}(\mathcal{B})
\end{aligned}$$

Next we provide our reduction to $\mathrm{G}^{\mathcal{G}}$ security for any $\mathcal{B}$ playing $\mathrm{G}^{\mathsf{mu}\text{-}\mathcal{G}}$ and making at most one IFING query. We assume without loss of generality that labels queried to IREG have the form $(\mathcal{I}, i)$ for $i \in [q_{\text{IREG}}]$ and that labels queries as the second input to IREQ have the form $(\mathcal{R}, j)$ for $j \in [q_{\text{IREQ}}]$. We assume $\mathcal{B}$ never makes queries which trigger "Require" statements (allowing us to omit "Require" statements from our code and assume that O and EXP are only appropriately called with the labels queried to IFING).

Our adversary $\mathcal{C}$ is shown in Fig. 3.11 and starts by guessing indices i and j corresponding to what labels will later be queries to IFING. If it guessed incorrectly it will output $b' = 0$. It uses its own game to simulate oracles for which these labels are queried. For all other labels it will simulate the ideal key exchange locally.

It additionally guesses a value $d \in \{0, 1, 2\}$. This is a guess whether $(\mathcal{R}, j)$ will be queried to IREVEAL first ($d = 2$), $(\mathcal{I}, i)$ will be queried to IREVEAL first ($d = 1$), or neither will be queried ($d = 0$). In the last case it simulated the shadow key returned by IREQ when queried for $(\mathcal{R}, j)$ by picking it at random. In the other cases it exposes the appropriate user in its own game and returns the obtained key as the shadow key. Note that guessing which user is revealed first is necessary so that $\mathcal{C}$'s transcript of exposes matches the one that would be induced for $\mathcal{B}$ in IFING. If $\mathcal{C}$ guessed the incorrect d then it simply outputs $b' = 0$ (using the pseudocode **abort**(0)).

When $\mathcal{C}$ correctly guesses i, j, and d it perfectly simulates the view of $\mathcal{B}$ playing $\mathrm{G}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi,b}$ when it is playing $\mathrm{G}^{\mathcal{G}}_{\Pi,b}$ so it outputs $b' = 1$ with the same probability as $\mathcal{A}$. Moreover its

transcript $\mathbf{T}$ perfectly matches $\mathbf{T}[\mathsf{lab}^*]$, so this b' is overwritten to 0 with the same probability. Its probability of guessing these values correctly is independent of b because the checks whether the guesses were correct occur before OS is ever queried. Thus the probability it did not guess these values incorrectly has probability at least $1/(3 \cdot q_{\mathrm{IReg}} \cdot q_{\mathrm{IReq}})$ so $\mathsf{Adv}^{\mathsf{mu}\text{-}\mathcal{G}}_{\Pi}(\mathcal{B}) \leq (3 \cdot q_{\mathrm{IReg}} \cdot q_{\mathrm{IReq}})\mathsf{Adv}^{\mathcal{G}}_{\Pi}(\mathcal{C})$. The lemma follows by combining our individual results. □

3.8.1 Proof of Theorem 15

We start our proof by constructing an adversary $\mathcal{A}_{\mathsf{KE}}$ against the security of KE. It will emulate the view of $\mathcal{A}_{\Pi,\mathsf{KE}}$ using its own oracles to emulate the key exchange. For when two session perform a successful fingerprint comparison the keys are returned to $\mathcal{A}_{\mathsf{KE}}$ which uses them to simulate an instance interacting with Π using $\mathcal{G}$ using a secret bit d_{Π}. If $\mathcal{A}_{\Pi,\mathsf{KE}}$ correctly guesses this bit, $\mathcal{A}_{\mathsf{KE}}$ outputs 1. Otherwise it outputs 0.

Adversary $\mathcal{A}_{\mathsf{KE}}$ is formally defined by the pseudocode in Fig. 3.12 and Fig. 3.13. Highlighting is used to indicate where we have changed the code of oracles from those in $\mathrm{G}^{\mathsf{oke}\text{-}\mathcal{G}}$. Most importantly, we check before each key exchange query whether it will make $\mathsf{V}(\mathbf{T}) = \mathsf{true}$ in which case we abort immediately. Additionally, for technical reasons that will become apparent later, in FINGS we check if the keys produced were $\perp$ rather than comparing fingerprints. This is equivalent. If the fingerprints are not equal the returned keys will be $\perp$. If the fingerprints are equal then the keys cannot be $\perp$ (because this would imply that the sessions do not have status $\mathsf{waiting}$). This would have caused V to reject the transcript from our minimal assumptions on V.

We note that the view of $\mathcal{A}_{\Pi,\mathsf{KE}}$ when run by $\mathcal{A}_{\mathsf{KE}}$ (when the secret bit of $\mathrm{G}^{\mathsf{oke}}$ is 1) is identical to its view in $\mathrm{G}^{\mathsf{oke}\text{-}\mathcal{G}}$ with secret bit d_{Π} (with the exception that we abort early when we already know $\mathsf{V}(\mathbf{T})$ would become true). The oracles it is given access to are identical between the two games. Additionally note that $\mathsf{V}(\mathbf{T})$ will never hold in $\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},1}$. Thus we can verify that $\mathcal{A}_{\mathsf{KE}}$ outputs 1 exactly when $b' = b$ would hold at the end of $\mathrm{G}^{\mathsf{oke}\text{-}\mathcal{G}}_{\Pi,\mathsf{KE},\mathcal{P},b}(\mathcal{A}_{\Pi,\mathsf{KE}})$. Thus, standard

Adversary $\mathcal{C}^{\mathrm{O,Exp}}(\ell)$	$\mathrm{OS}(x, \mathsf{lab})$
$i \leftarrow\$ [q_{\mathrm{IReg}}]$; $j \leftarrow\$ [q_{\mathrm{IReq}}]$ $d \leftarrow\$ \{0,1,2\}$ $\mathsf{xp} \leftarrow \mathsf{true}$ $(pk_{(\mathcal{I},i)}, pk_{(\mathcal{R},j)}) \leftarrow \ell$ $b' \leftarrow\$ \mathcal{B}^{\mathcal{O}\mathrm{S,OS,ExpS}}$ Return b'	Return $\mathrm{O}(x)$ $\mathrm{ExpS}(\mathsf{lab})$ If $\mathsf{lab} = (\mathcal{I}, i)$ Return $\mathrm{Exp}(\mathcal{I})$ Return $\mathrm{Exp}(\mathcal{R})$

$\mathrm{IFingS}(\mathsf{lab}_1, \mathsf{lab}_2)$
If $\{\mathsf{lab}_1, \mathsf{lab}_2\} \neq \{(\mathcal{I},i), (\mathcal{R},j)\}$ then **abort**(0)
If $\{\mathsf{lab}_1, \mathsf{lab}_2\} \cap \mathcal{X} = \emptyset$ and $d \neq 0$ then **abort**(0)
Return ε

$\mathrm{IRegS}(\mathsf{lab})$	$\mathrm{IFinS}(\mathsf{lab}_1, \mathsf{lab}_2)$

If $\mathsf{lab} = (\mathcal{I}, i)$ then $(pk_{\mathcal{I}}, sk_{\mathcal{I}}) \leftarrow (pk_{(\mathcal{I},i)}, \diamond)$
Else $(pk_{\mathcal{I}}, sk_{\mathcal{I}}) \leftarrow\$ \Pi.\mathsf{Kg}_{\mathcal{I}}$
$\vec{k}[\mathsf{lab}] \leftarrow ((pk_{\mathcal{I}}, \perp), sk_{\mathcal{I}}, \perp)$
$\vec{k}'[\mathsf{lab}] \leftarrow ((pk_{\mathcal{I}}, \perp), sk_{\mathcal{I}}, \perp)$
Return $pk_{\mathcal{I}}$
 	
$\mathrm{IReqS}(\mathsf{lab}_1, \mathsf{lab}_2)$
If $\mathsf{lab}_2 = (\mathcal{R}, j)$ and $\mathsf{lab}_1 \neq (\mathcal{I}, i)$ then
  abort(0)
If $\mathsf{lab}_2 = (\mathcal{R}, j)$ then
  $(pk_{\mathcal{R}}, sk_{\mathcal{R}}) \leftarrow (pk_{(\mathcal{R},j)}, \diamond)$
  If $d = 0$ then $k_{\mathcal{R}} \leftarrow \diamond$; $k'_{\mathcal{R}} \leftarrow\$ \{0,1\}^{\Pi.\mathsf{k}}$
  If $d = 1$ then
    If $k^* = \perp$ then $(\cdot, sk_{(\mathcal{I},i)}, k^*) \leftarrow \mathrm{Exp}(\mathcal{I})$
    $k_{\mathcal{R}} \leftarrow\$ \{0,1\}^{\Pi.\mathsf{k}}$; $k'_{\mathcal{R}} \leftarrow k^*$
  If $d = 2$ then
    $k_{\mathcal{R}} \leftarrow\$ \{0,1\}^{\Pi.\mathsf{k}}$; $(\cdot, sk_{(\mathcal{R},j)}, k'_{\mathcal{R}}) \leftarrow$
$\mathrm{Exp}(\mathcal{R})$
Else
  $(pk_{\mathcal{R}}, sk_{\mathcal{R}}) \leftarrow\$ \Pi.\mathsf{Kg}_{\mathcal{R}}$
  $k_{\mathcal{R}} \leftarrow\$ \{0,1\}^{\Pi.\mathsf{k}}$; $k'_{\mathcal{R}} \leftarrow\$ \{0,1\}^{\Pi.\mathsf{k}}$
$((pk_{\mathcal{I}}, \cdot), \cdot, \cdot) \leftarrow \vec{k}[\mathsf{lab}_1]$; $\ell \leftarrow (pk_{\mathcal{I}}, pk_{\mathcal{R}})$
$\vec{k}[\mathsf{lab}_2] \leftarrow (\ell, sk_{\mathcal{R}}, k_{\mathcal{R}})$
$\vec{k}'[\mathsf{lab}_2] \leftarrow (\ell, sk_{\mathcal{R}}, k'_{\mathcal{R}})$
Return $(pk_{\mathcal{R}}, k'_{\mathcal{R}})$
	//Unchanged from IFin in $\mathrm{G}^{\mathsf{mu}\text{-}\mathcal{G}}$
	
	$\mathrm{IRevealS}(\mathsf{lab})$
	If $\mathsf{lab} \in \{(\mathcal{I},i), (\mathcal{R},j)\}$ and $d = 0$ then
	  abort(0)
	If $\mathsf{lab} = (\mathcal{I}, i)$ and $d = 2$ and xp then
	  abort(0)
	If $\mathsf{lab} = (\mathcal{R}, j)$ and $d = 1$ and xp then
	  abort(0)
	If $\mathsf{lab} = (\mathcal{I}, i)$ then
	  If xp then
	    $\mathsf{xp} \leftarrow \mathsf{false}$
	    If $k^* = \perp$ then $(\cdot, sk_{(\mathcal{I},i)}, k^*) \leftarrow \mathrm{Exp}(\mathcal{I})$
	  Else $(\cdot, sk_{(\mathcal{I},i)}, \cdot) \leftarrow \mathrm{Exp}(\mathcal{I})$
	If $\mathsf{lab} = (\mathcal{R}, j)$ then
	  If xp then $\mathsf{xp} \leftarrow \mathsf{false}$
	  Else $(\cdot, sk_{(\mathcal{R},j)}, \cdot) \leftarrow \mathrm{Exp}(\mathcal{R})$
	$(\ell, sk, k) \leftarrow \vec{k}'[\mathsf{lab}]$
	If $sk = \diamond$ then $\vec{k}'[\mathsf{lab}] \leftarrow (\ell, sk_{\mathsf{lab}}, k)$
	$\mathcal{X}.\mathsf{add}(\mathsf{lab})$; Return $\vec{k}'[\mathsf{lab}]$

Figure 3.11. Single user adversary for proof of Lemma 16. For compactness we let $\mathcal{O}\mathrm{S} = \mathrm{IRegS}, \mathrm{IReqS}, \mathrm{IFinS}, \mathrm{IRevealS}, \mathrm{IFingS}$.

$\mathcal{A}_{\mathsf{KE}}^{\text{REG,REQ,FIN,FING,CORRUPT,REVEAL}}$	$\text{OS}(x,u,i)$

$d_\Pi \leftarrow_\$ \{0,1\}$
$b' \leftarrow_\$ \mathcal{A}_{\Pi,\mathsf{KE}}^{\mathcal{K}\text{S},\text{EXPS},\text{OS}}$
For $\mathsf{lab} \in \mathcal{U}$ do
  If $\mathcal{G}.\mathbf{Pred}(\mathbf{T}[\mathsf{lab}])$
    $b' \leftarrow 0$
If $b' = d_\Pi$ then return 1
Return 0
	//Identical to O in $\text{G}^{\text{oke-}\mathcal{G}}$
	$\text{EXPS}(u,i)$
	//Identical to EXP in $\text{G}^{\text{oke-}\mathcal{G}}$

$\text{FINGS}(u,i,v,j)$

If $\mathsf{V}(\mathbf{T},\text{FING},(u,i,v,j))$
    If $d_\Pi = 1$ then **abort**(0)
    Else **abort**(1)
$(\vec{k}_1,\vec{k}_2) \leftarrow \text{FING}(u,i,v,j)$
If $(\vec{k}_1,\vec{k}_2) \neq (\perp,\perp)$ then
  $sst_u^i.\mathsf{status} \leftarrow \mathsf{accepted}$; $sst_v^j.\mathsf{status} \leftarrow \mathsf{accepted}$
  $sst_u^i.\mathsf{psess} \leftarrow j$; $sst_v^j.\mathsf{psess} \leftarrow i$
  $(\vec{k}[(u,i)],\vec{k}[(v,j)]) \leftarrow (\vec{k}_1,\vec{k}_2)$
  If $sst_u^i.\mathsf{role} = \mathcal{I}$ then $\mathsf{lab}^* \leftarrow (u,i)$
  Else $\mathsf{lab}^* \leftarrow (v,j)$
  $\mathcal{U} \leftarrow \mathcal{U} \cup \{\mathsf{lab}^*\}$; $\sigma_{\mathcal{G}}[\mathsf{lab}^*] \leftarrow_\$ \mathcal{G}.\mathbf{Start}(d_\Pi)$; $t \leftarrow \mathsf{I}(\mathbf{T},u,i,v,j)$
  For $(w,k) \in t$ do $\mathbf{T}[\mathsf{lab}^*].\mathsf{add}(\text{EXP}, sst_w^k.\mathsf{role}, \vec{k}[(w,k)])$
Else
  $sst_u^i.\mathsf{status} \leftarrow \mathsf{rejected}$; $sst_v^j.\mathsf{status} \leftarrow \mathsf{rejected}$
$\mathbf{T}.\mathsf{add}(\text{FING},(u,i,v,j),\varepsilon)$; Return ε

Figure 3.12. Adversary $\mathcal{A}_{\mathsf{KE}}$ for proof of Theorem 15. For compactness we define $\mathcal{K}\text{S} = \text{REGS}, \text{REQS}, \text{FINS}, \text{FINGS}, \text{CORRUPTS}, \text{REVEALS}$.

---

probability rewriting gives $\mathsf{Adv}_{\Pi,\mathsf{KE},\mathcal{P}}^{\mathsf{oke}\text{-}\mathcal{G}}(\mathcal{A}_{\Pi,\mathsf{KE}}) = 2\Pr[\text{G}_{\mathsf{KE},\mathcal{P},\mathsf{S},1}^{\mathsf{oke}}(\mathcal{A}_{\mathsf{KE}})] - 1$.

Next we introduce adversary $\mathcal{A}_\Pi$ against the security of Π. Specifically it is playing $\text{G}_{\Pi,b}^{\mathsf{mu}\text{-}\mathcal{G}}(\mathcal{A})$. It simulates the key exchange queries of $\mathcal{A}_{\Pi,\mathsf{KE}}$ by running S with access to its own ideal key exchange oracles. Then any queries to O or EXP by $\mathcal{A}_{\Pi,\mathsf{KE}}$ are forwarded to its own oracles. The goal of $\mathcal{A}_\Pi$ is that the view of $\mathcal{A}_\Pi$ matches what it would see in $\text{G}_{\mathsf{KE},\mathcal{P},\mathsf{S},0}^{\mathsf{oke}}$ when run by $\mathcal{A}_{\mathsf{KE}}$ with the secret bit $\mathcal{A}_\Pi$ is trying to guess playing the role of d_Π.

The adversary is formally given in Fig. 3.14 and Fig. **??**. We have used highlighting to indicate where we have made changes from existing oracles. In particular, the IFING which $\mathcal{A}_\Pi$ has access to does not return keys (unlike the one used in $\text{G}_{\mathsf{KE},\mathcal{P},\mathsf{S},0}^{\mathsf{oke}}$). Instead $\mathcal{A}_\Pi$ simply queries

$\underline{\text{RegS}(u)}$
If $\mathsf{V}(\mathbf{T}, \text{Req}, u)$
  If $d_\Pi = 1$ then **abort**(0)
  Else **abort**(1)
$m \leftarrow \text{Reg}(u)$
$\mathbf{T}.\mathsf{add}(\text{Reg}, u, m)$
Return m

$\underline{\text{FinS}(u, i, m, v)}$
If $\mathsf{V}(\mathbf{T}, \text{Fin}, (u, i, m, v))$
  If $d_\Pi = 1$ then **abort**(0)
  Else **abort**(1)
$fp \leftarrow \text{Fin}(u, i, m, v)$
$sst_u^i.\mathsf{fing} \leftarrow fp$
$sst_u^i.\mathsf{role} \leftarrow \mathcal{I}$
$sst_u^i.\mathsf{pid} \leftarrow v$
If $fp = \perp$ then
  $sst_u^i.\mathsf{status} \leftarrow \mathsf{rejected}$
Else
  $sst_u^i.\mathsf{status} \leftarrow \mathsf{waiting}$
$\mathbf{T}.\mathsf{add}(\text{Fin}, (u, i, m, v), fp)$
Return fp

$\underline{\text{ReqS}(u, i, m, v)}$
If $\mathsf{V}(\mathbf{T}, \text{Req}, (u, i, m, v))$
  If $d_\Pi = 1$ then **abort**(0)
  Else **abort**(1)
$(m', fp) \leftarrow \text{Req}(u, i, m, v)$
$sst_u^i.\mathsf{fing} \leftarrow fp$
$sst_u^i.\mathsf{role} \leftarrow \mathcal{R}$
$sst_u^i.\mathsf{pid} \leftarrow v$
If $fp = \perp$ then
  $sst_u^i.\mathsf{status} \leftarrow \mathsf{rejected}$
Else
  $sst_u^i.\mathsf{status} \leftarrow \mathsf{waiting}$
$\mathbf{T}.\mathsf{add}(\text{Req}, (u, i, m, v), (m', fp))$
Return (m', fp)

$\underline{\text{CorruptS}(u)}$
If $\mathsf{V}(\mathbf{T}, \text{Corrupt}, u)$
  If $d_\Pi = 1$ then **abort**(0)
  Else **abort**(1)
$st_u \leftarrow \text{Corrupt}(u)$
$\mathbf{T}.\mathsf{add}(\text{Corrupt}, u, st_u)$
Return st_u

$\underline{\text{RevealS}(u, i)}$
If $\mathsf{V}(\mathbf{T}, \text{Reveal}, (u, i))$
  If $d_\Pi = 1$ then **abort**(0)
  Else **abort**(1)
$sst_u^i.\mathsf{keys} \leftarrow \text{Reveal}(u, i)$
$\mathbf{T}.\mathsf{add}(\text{Reveal}, (u, i), sst_u^i.\mathsf{keys})$
Return $sst_u^i.\mathsf{keys}$

Figure 3.13. Additional oracles as simulated by $\mathcal{A}_{\mathsf{KE}}$.

IFing and remembers the association between sessions and labels that it will need to access these keys in future queries to O or Exp.

Now the claim that we need to justify to complete our proof is that $\mathsf{Adv}_{\Pi}^{\mathsf{mu}\text{-}\mathcal{G}}(\mathcal{A}_\Pi) = 2\Pr[\mathsf{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},0}(\mathcal{A}_{\mathsf{KE}})] - 1$ which will follow by showing that arguing that $\mathcal{A}_\Pi$ achieves its goal.

```
Adversary A_Π^{O,IFing,O,Exp}
σ ←$ S(Init)
b' ←$ A_{Π,KE}^{KS,ExpS,OS}
If V(T) then b' ← 0
Return b'

OS(x,u,i)
Return O(z, lab[(u,i)])

ExpS(u,i)
Return Exp(lab[(u,i)])

FingS(u,i,v,j)
(σ, y, lab_1, lab_2) ←$ S^O(σ, Fing, u, i, v, j)
t ← I(T, u, i, v, j) ; t' ← []
For lab ∈ X do
   If lab = lab_1 then t'.add((u,i))
   If lab = lab_2 then t'.add((v,j))
If y and t = t' then
   x ← IFing(lab_1, lab_2)
   If x ≠ ⊥ then (lab[(u,i)], lab[(v,j)]) ← (lab_1, lab_2)
T.add(Fing, (u,i,v,j), ε) ; Return ε

RegS, ReqS, FinS, CorruptS, RevealS
//Identical to corresponding oracle in G^{oke}_{KE,P,S,0}
```

Figure 3.14. Adversary $\mathcal{A}_\Pi$ for proof of Theorem 15. For compactness we define $\mathcal{K}\mathrm{S} = \mathrm{RegS}$, $\mathrm{ReqS}, \mathrm{FinS}, \mathrm{FingS}, \mathrm{CorruptS}, \mathrm{RevealS}$. Additionally, we define $\mathcal{O} = \mathrm{IReg}, \mathrm{IReq}$, $\mathrm{IFin}, \mathrm{IReveal}$.

This allows us to compute the following.

$$\begin{aligned}
\mathsf{Adv}^{\text{oke-}\mathcal{G}}_{\Pi,\mathsf{KE},\mathcal{P}}(\mathcal{A}_{\Pi,\mathsf{KE}}) &= 2\Pr[\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},1}(\mathcal{A}_{\mathsf{KE}})] - 1 \\
&= 2(\mathsf{Adv}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S}}(\mathcal{A}_{\mathsf{KE}}) + \Pr[\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},0}(\mathcal{A}_{\mathsf{KE}})]) - 1 \\
&= 2\mathsf{Adv}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S}}(\mathcal{A}_{\mathsf{KE}}) + (2\Pr[\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},0}(\mathcal{A}_{\mathsf{KE}})] - 1) \\
&= 2\mathsf{Adv}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S}}(\mathcal{A}_{\mathsf{KE}}) + \mathsf{Adv}^{\text{mu-}\mathcal{G}}_{\Pi}(\mathcal{A}_\Pi)
\end{aligned}$$

The final theorem would then follow by applying Lemma 16.

We complete the proof by arguing that the view of $\mathcal{A}_\Pi$ matches what it would see in $\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},0}$ when run by $\mathcal{A}_{\mathsf{KE}}$ with the secret bit $\mathcal{A}_\Pi$ is trying to guess playing the role of d_Π. In the latter case $\mathcal{A}_{\mathsf{KE}}$ will abort early as soon as $\mathsf{V}(\mathbf{T})$ would hold instead of waiting to check it at the end, but clearly this does not effect the probabilities involved. Other than that the key

exchange oracle simulations (other than that of FING) essentially just give $\mathcal{A}_{\Pi,\mathsf{KE}}$ direct access to the corresponding oracle. Since $\mathcal{A}_{\Pi}$ leaves these unchanged from $\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},0}$, this simulation is clearly correct.

Now we consider whether fingerprinting correctly links the key exchange oracles with the messaging oracles. Suppose the keys returned to $\mathcal{A}_{\mathsf{KE}}$ would be $\perp$ in its FINGS (causing it not to store them and make them available via the messaging oracles). Then in FING_0 either $y = \mathsf{false}$ or $t \neq t'$ or one of IFING's "Require" statements was triggered. In any of these cases $\mathcal{A}_{\Pi}$ would not save the labels for the session so they would be similarly be inaccessible through the messaging oracles. In the other case, $\mathcal{A}_{\mathsf{KE}}$'s computation of lab^* mirrors that of $\mathcal{A}_{\Pi}$'s oracle IFING (because $sst_u^i.\mathsf{role}$ computed by $\mathcal{A}_{\mathsf{KE}}$ matches $mst[\mathsf{lab}[(u,i)]].\mathsf{role}$ computed by IFING) so they will be similarly accessible in the same manner through the messaging oracles. Note, moreover that the requirement $t = t'$ implies that the $\mathbf{T}[\mathsf{lab}^*]$ computed by $\mathcal{A}_{\mathsf{KE}}$ matches the one computed in IFING. Since $\mathcal{A}_{\mathsf{KE}}$ outputs 1 whenever $\mathcal{A}_{\Pi,\mathsf{KE}}$ has correctly guessed d_{Π} the claim mentioned earlier holds, completing the proof.

3.9 Security of GKE

In this section we formalize and prove the security of GKE. This is captured by the following theorem which reduces its security to that of PKE and H.

Theorem 17. *Let* $\mathsf{GKE} = \mathsf{GKE}[\mathsf{PKE}, n]$ *where* PKE *is a public key encryption scheme,* $n \in \mathbb{N}$*, and* GKE.Init *is a splittable initialization algorithm. Let* $\mathcal{P} = (\mathsf{I}^*, \mathsf{V}^*)$*. Let* S *be the simulator specified in Appendix 3.6 and* $\mathcal{A}$ *be an adversary making at most* q_{X} *queries to each of its oracles* X*. Then we can construct adversaries* $\mathcal{P}$ *and* $\mathcal{H}$ *satisfying*

$$\mathsf{Adv}^{\mathsf{oke}}_{\mathsf{GKE},\mathcal{P},\mathsf{S}}(\mathcal{A}) \leq 2q \cdot \mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE}}(\mathcal{P}) + \mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathcal{H}) + nq^2 2^{-\mathrm{H}_\infty(\mathsf{PKE})}.$$

Here q *is the sum of all of the* q_{X}*'s. These adversaries are comparably efficient to* $\mathcal{A}$.

The rest of this section proves this result. We start with a useful lemma and then proceed to the main body of the proof.

Public Key Encryption Entropy.

In the theorem above we use the min-entropy of the public key encryption scheme PKE. We first define the min-entropy of algorithms $\mathsf{PKE.Kg}$ and $\mathsf{PKE.Enc}$ as $\mathrm{H}_\infty(\mathsf{PKE.Kg})$ and $\mathrm{H}_\infty(\mathsf{PKE.Enc})$, such that

$$2^{-\mathrm{H}_\infty(\mathsf{PKE.Kg})} = \max_{ek \in \{0,1\}^*} \Pr[ek^* = ek : (ek, \cdot) \leftarrow\!\!\$\, \mathsf{PKE.Kg}]$$

$$2^{-\mathrm{H}_\infty(\mathsf{PKE.Enc})} = \max_{c, ek, m \in \{0,1\}^*} \Pr[c^* = c : c \leftarrow\!\!\$\, \mathsf{PKE.Enc}(ek, m)].$$

Note that the entropy of key generation only depends on ek, not dk. Then we define $\mathrm{H}_\infty(\mathsf{PKE})$ to be the minimum of $\mathrm{H}_\infty(\mathsf{PKE.Kg})$ and $\mathrm{H}_\infty(\mathsf{PKE.Enc})$. We note that the entropy of PKE can be augmented by adding additionally random bit to ek and c which are ignored by all algorithms.

Public Key Encryption Lemma.

We start our proof by introducing a new security definition for public key encryption. We do not expect this definition to be of independent interest, but instead introduce it for the sake of modularity and understandability of our ultimate proof. In particular it precisely captures the use-case of the public key encryption scheme that is required to prove security of our proposed key exchange protocol.

Consider the game $\mathrm{G}^{\mathsf{mu-pke}}_{\mathsf{PKE},\kappa,b}$ shown in Fig. 3.15. It is parameterized by a public key encryption scheme PKE, key length $\kappa \in \mathbb{N}$, and bit b which the adversary attempts to guess. The advantage of an adversary $\mathcal{A}$ is defined by $\mathsf{Adv}^{\mathsf{mu\text{-}pke}}_{\mathsf{PKE},\kappa}(\mathcal{A}) = \Pr[\mathrm{G}^{\mathsf{mu\text{-}pke}}_{\mathsf{PKE},\kappa,1}(\mathcal{A})] - \Pr[\mathrm{G}^{\mathsf{mu\text{-}pke}}_{\mathsf{PKE},\kappa,0}(\mathcal{A})]$.

In this game, the adversary may interact with an arbitrary number of users i by calling $\textsc{New}$ with input i. When it does so it receives back the encryption key of i. Once i is created it may request as many new ciphertexts as it wishes from that user by calling $\textsc{Enc}(i, j)$. The ciphertext will encrypt $k^1_{i,j}$ which was just sampled uniformly at random.

Game $\mathrm{G}^{\text{mu-pke}}_{\mathsf{PKE},\kappa,b}(\mathcal{A})$	$\underline{\textsc{Show}(i)}$
$b' \leftarrow\$ \mathcal{A}^{\textsc{New},\textsc{Enc},\textsc{Show},\textsc{Chal},\textsc{Key}}$	Require $(\textsc{New}, i) \in S$
Return $(b' = 1)$	Require $\forall j, (\textsc{Chal}, (i, j)) \notin S$
$\underline{\textsc{New}(i)}$	$S.\mathsf{add}(\textsc{Show}, i)$
Require $(\textsc{New}, i) \notin S$	Return dk_i
$(ek_i, dk_i) \leftarrow\$ \mathsf{PKE.Kg}$	$\underline{\textsc{Chal}(i, j)}$
$S.\mathsf{add}(\textsc{New}, i)$	Require $(\textsc{Enc}, (i, j)) \in S$
Return ek_i	Require $\forall j', (\textsc{Chal}, (i, j')) \notin S$
$\underline{\textsc{Enc}(i, j)}$	Require $(\textsc{Show}, i) \notin S$
Require $(\textsc{New}, i) \in S$	Require $(\textsc{Key}, (i, j) \notin S$
Require $(\textsc{Enc}, (i, j)) \notin S$	$k^0_{i,j} \leftarrow\$ \{0, 1\}^\kappa$
$k^1_{i,j} \leftarrow\$ \{0, 1\}^\kappa$	$S.\mathsf{add}(\textsc{Chal}, (i, j))$
$c_{i,j} \leftarrow\$ \mathsf{PKE.Enc}(ek_i, k^1_{i,j})$	Return $k^b_{i,j}$
$S.\mathsf{add}(\textsc{Enc}, (i, j))$	$\underline{\textsc{Key}(i, j)}$
Return $c_{i,j}$	Require $(\textsc{Enc}, (i, j)) \in S$
	Require $(\textsc{Chal}, (i, j)) \notin S$
	$S.\mathsf{add}(\textsc{Key}, (i, j))$
	Return $k^1_{i,j}$

Figure 3.15. Security game for public key encryption.

Then the attacker can call either $\textsc{Show}(i)$, $\textsc{Chal}(i, j)$, or $\textsc{Key}$. But for a particular i it may only may call $\textsc{Chal}$ for a single value of j and may not call either of the other two oracles if it does so. In $\textsc{Show}$, the decryption key dk_i is returned to $\mathcal{A}$ while in $\textsc{Key}$ the key $k^1_{i,j}$ is returned to $\mathcal{A}$. Either of these allow the adversary to learn what was encrypted in $c_{i,j}$ but intuitively do not help it guess b because its view only depend on the secret bit when it makes queries to $\textsc{Chal}$ this oracle either returns $k^1_{i,j}$ or a fresh random key $k^0_{i,j}$ depending on the value of the secret bit. Note that "Require" statements are used to capture the various restrictions we stated on the behavior of $\mathcal{A}$.

The following lemma exhibits that security with respect to this security notion is implied by security with respect to standard semantic security of a public key encryption scheme. To prove this lemma we first use a hybrid argument to reduce to bounding the security of an adversary making only one query to $\textsc{New}$. We can assume such an adversary never queries $\textsc{Show}$ which allows us to apply the security of PKE to replace the encryption of k^1 with the

Adversary $\mathcal{B}^{\text{New,Enc,Show,Chal,Key}}$	$\text{ShowS}(i)$
$i^* \leftarrow\$ [q_{\text{New}}]$	If $i = i^*$
$b' \leftarrow\$ \mathcal{A}^{\text{NewS,EncS,ShowS,ChalS,KeyS}}$	$\quad dk_i \leftarrow \text{Show}(i)$
Return b'	Return dk_i
$\text{NewS}(i)$	$\text{ChalS}(i,j)$
If $i = i^*$	If $i = i^*$
$\quad ek_i \leftarrow \text{New}(i)$	$\quad k \leftarrow \text{Chal}(i,j)$
Else	Else if $i > i^*$
$\quad (ek_i, dk_i) \leftarrow\$ \mathsf{PKE.Kg}$	$\quad k \leftarrow\$ \{0,1\}^\kappa$
Return ek_i	Else if $i < i^*$
$\text{EncS}(i,j)$	$\quad k \leftarrow k^1_{i,j}$
If $i = i^*$	Return k
$\quad c_{i,j} \leftarrow \text{Enc}(i,j)$	$\text{KeyS}(i,j)$
Else	If $i = i^*$
$\quad k^1_{i,j} \leftarrow\$ \{0,1\}^\kappa$	$\quad k^1_{i,j} \leftarrow \text{Key}(i,j)$
$\quad c_{i,j} \leftarrow\$ \mathsf{PKE.Enc}(ek_i, k^1_{i,j})$	Return $k^1_{i,j}$
Return $c_{i,j}$	

Figure 3.16. Adversary $\mathcal{B}$ used for proof of Lemma 18. For compactness, "Require" statements are omitted.

---

encryption of 0^κ which makes the adversaries view independent of the bit it is trying to guess.

Lemma 18. *Let* PKE *be a public key encryption scheme for messages of length* $\kappa \in \mathbb{N}$*. Let* $\mathcal{A}$ *be an adversary making at most* q *queries to its* New *oracle. Then we can create adversary* $\mathcal{P}$ *for which the following holds.*

$$\mathsf{Adv}^{\mathsf{mu\text{-}pke}}_{\mathsf{PKE},\kappa}(\mathcal{A}) \leq 2q \cdot \mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE}}(\mathcal{P})$$

Adversary $\mathcal{P}$ *is comparably efficient to* $\mathcal{A}$*.*

Proof. Without loss of generality we can assume that $\mathcal{A}$ never makes queries which would trigger a "Require" statement and only makes queries with $i \in [q]$. We start our proof with a multi-user to single user hybrid argument so that we only need to consider adversaries which only make a single to Enc.

Adversary $\mathcal{P}^{\mathrm{LR}}(ek)$	$\mathrm{SHOWS}(i,j)$
$d_{\mathsf{pke}} \leftarrow\!\$ \{0,1\}$	Return $\diamond$
$b' \leftarrow\!\$ \mathcal{A}^{\mathrm{NEWS,ENCS,SHOWS,CHALS,KEYS}}$	$\mathrm{CHALS}(i)$
If $b' = d$ then return 1	$k^0_{i,j} \leftarrow\!\$ \{0,1\}^*$
Else return 0	Return $k^d_{i,j}$
$\mathrm{NEWS}(i)$	$\mathrm{KEYS}(i,j)$
Return ek	Return $k^1_{i,j}$
$\mathrm{ENCS}(i,j)$	
$k^1_{i,j} \leftarrow\!\$ \{0,1\}^\kappa$	
$c_{i,j} \leftarrow \mathrm{LR}(0^\kappa, k^1_{i,j})$	
Return $c_{i,j}$	

Figure 3.17. Adversary $\mathcal{P}$ used for proof of Lemma 18. For compactness, "Require" statements are omitted.

Consider the adversary $\mathcal{B}$ shown in Fig. 3.16. It starts by sampling a random $i^* \in [q]$. All queries made for $i = i^*$ are simply forwarded to $\mathcal{B}$'s own oracles. It simulates the responses to all other queries locally. When $i > i^*$ it returns a random key in CHALS (emulating $k^0_{i,j}$) and when $i < i^*$ it return $k^1_{i,j}$.

Let $\Pr[\mathcal{B}^b_i]$ denote the probability that $\mathcal{B}$ outputs $b' = 1$ conditioned on it having sampled $i^* = i$ and the underlying bit in $\mathrm{G}^{\mathsf{mu\text{-}pke}}$ is b. Thus $\mathsf{Adv}^{\mathsf{mu\text{-}pke}}_{\mathsf{PKE},\kappa}(\mathcal{B}) = \mathbf{E}_i(\Pr[\mathcal{B}^1_i]) - \mathbf{E}_i(\Pr[\mathcal{B}^0_i])$. We make the following three claims.

1. $\Pr[\mathcal{B}^1_{q-1}] = \Pr[\mathrm{G}^{\mathsf{mu\text{-}pke}}_{\mathsf{PKE},\kappa,1}(\mathcal{A})]$

2. $\Pr[\mathcal{B}^0_0] = \Pr[\mathrm{G}^{\mathsf{mu\text{-}pke}}_{\mathsf{PKE},\kappa,0}(\mathcal{A})]$

3. $\Pr[\mathcal{B}^1_x] = \Pr[\mathcal{B}^0_{x+1}]$ for all x

For claim (1) note that when $i^* = q-1$ it will never hold that $i > i^*$. This and $b = 1$ implies $k^1_{i,j}$ is always being returned to $\mathcal{A}$ by CHALS as in $\mathrm{G}^{\mathsf{mu\text{-}pke}}_{\mathsf{PKE},\kappa,1}$. For claim (2) note that when $i^* = 0$ it will never hold that $i < i^*$. This and $b = 0$ implies a fresh random key is always being returned to $\mathcal{A}$ by CHALS as in $\mathrm{G}^{\mathsf{mu\text{-}pke}}_{\mathsf{PKE},\kappa,0}$. For claim (3) note that in both of these cases $k^1_{i,j}$ is being returned to $\mathcal{A}$ by CHALS when $i \leq x$ and otherwise a fresh random key is being returned.

The claims give us the following.

$$\begin{aligned}\mathsf{Adv}^{\mathsf{mu\text{-}pke}}_{\mathsf{PKE},\kappa}(\mathcal{A}) &= \Pr[\mathcal{B}^1_{q-1}] - \Pr[\mathcal{B}^0_0] \\ &= \Pr[\mathcal{B}^1_{q-1}] - \Pr[\mathcal{B}^0_0] + \sum_{x=0}^{q-2}(\Pr[\mathcal{B}^1_x] - \Pr[\mathcal{B}^0_{x+1}]) \\ &= \sum_{x=0}^{q-1}(\Pr[\mathcal{B}^1_x] - \Pr[\mathcal{B}^0_x]) = q\cdot \mathsf{Adv}^{\mathsf{mu\text{-}pke}}_{\mathsf{PKE},\kappa}(\mathcal{B})\end{aligned}$$

Note that because $\mathcal{B}$ only ever makes one NEW query, if it ever calls SHOW then it can no longer call CHAL and its view is independent of the bit it is trying to guess. So we can without loss of generality assume that it never does so. Let G'_b be defined identically to $\mathrm{G}^{\mathsf{mu\text{-}pke}}_{\mathsf{PKE},\kappa,1}(\mathcal{B})$ except ENC is redefined as follows.

$\underline{\mathrm{ENC}(i,j)}$

Require $(\mathrm{NEW}, i) \in S$

Require $(\mathrm{ENC}, (i,j)) \notin S$

$k^1_{i,j} \leftarrow\$ \{0,1\}^\kappa$

$c_{i,j} \leftarrow\$ \mathsf{PKE.Enc}(ek_i, 0^\kappa)$

$S.\mathsf{add}(\mathrm{ENC}, (i,j))$

Return $c_{i,j}$

The only difference is that 0^κ is encrypted instead of $k^1_{i,j}$. Because the ciphertext is now independent of $k^1_{i,j}$, we can see that $\Pr[\mathrm{G}'_1] - \Pr[\mathrm{G}'_0] = 0$. Then we construct adversary $\mathcal{P}$ as shown in Fig. 3.17. It samples its own bit d and simulates the view of $\mathcal{B}$ using its LR oracle to encrypt either $k^1_{i,j}$ or 0^κ then guesses 1 if $\mathcal{B}$ correctly guesses d. Standard conditional probability calculation then give that $\mathsf{Adv}^{\mathsf{mu\text{-}pke}}_{\mathsf{PKE},\kappa}(\mathcal{B}) = 2\cdot\mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE}}(\mathcal{P})$. The lemma follows by combining our individual results. □

3.9.1 Proof of Theorem 17

Our proof will consist of a sequence of game transitions bridging the gap between games $\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},1}$ and $\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},0}$. In particular we define game G_x for $x \in \{1,\dots,4\}$ each of the form specified below which differ only in the definition of the oracles. Throughout we use our observation from Section 3.7 that V^* is monotonic and efficiently computable in an online manner so we can assume that $\mathcal{A}$ never makes oracle queries which would cause V^* to reject the produced transcript.

$\underline{\mathrm{G}_x}$
$\sigma \leftarrow_{\$} \mathsf{S}(\textsc{Init})$
$b' \leftarrow_{\$} \mathcal{A}^{\textsc{Reg}_x,\textsc{Req}_x,\textsc{Fin}_x,\textsc{Fing}_x,\textsc{Corrupt}_x,\textsc{Reveal}_x}$
If $\mathsf{V}(\mathbf{T})$ then $b' \leftarrow 0$
Return $(b' = 1)$

Of these games we will prove the following results.

1. $\Pr[\mathrm{G}_0] = \Pr[\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},0}]$
2. $\Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_0] \leq 2q \cdot \mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE}}(\mathcal{P})$
3. $\Pr[\mathrm{G}_2] = \Pr[\mathrm{G}_1]$
4. $\Pr[\mathrm{G}_3] - \Pr[\mathrm{G}_2] \leq nq^2 \cdot 2^{-\mathrm{H}_\infty(\mathsf{PKE})}$
5. $\Pr[\mathrm{G}_4] - \Pr[\mathrm{G}_3] \leq \mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathcal{H})$
6. $\Pr[\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},1}] = \Pr[\mathrm{G}_4]$

The final claim is then obtained by the following calculation.

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S}}(\mathcal{A}) &= \Pr[\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},1}] - \Pr[\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},0}] \\
&= \Pr[\mathrm{G}_4] - \Pr[\mathrm{G}_0] \\
&= \sum_{x=1}^{4} \Pr[\mathrm{G}_x] - \Pr[\mathrm{G}_{x-1}] \\
&\leq 2q \cdot \mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE}}(\mathcal{P}) + nq^2 \cdot 2^{-\mathrm{H}_\infty(\mathsf{PKE})} + \mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathcal{H})
\end{aligned}$$

The brunt of the proof consists of the three arguments in Claims (2), (4), and (5). In Claim (4) we use the security of PKE to argue that $\mathcal{A}$ cannot tell the shadow key is being encrypted instead of the real key. In Claim (4) we use the entropy of PKE to argue that two sessions will only have the same view of the transcript of their respective conversations if $\mathcal{A}$ was, in fact, forwarding the messages between them honestly. Finally, in Claim (5) collision resistance of H is used to argue that S's use of the transcript for fingerprint comparisons is indistinguishable from the actual protocols use of the hash of the transcript. The rest of the claims follow from the fact that the various intermediate games were created from each other by making transforms to the pseudocode which we can argue do not change the behavior of the game.

Claim 1, $\Pr[\mathrm{G}_0] = \Pr[\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},0}]$**.**

Game G_0 is defined by the oracles in Figures 3.18 and 3.19. Note that the boxed code is not included in G_0. This game was created by hardcoding the pseudocode of S into $\mathrm{G}^{\mathsf{oke}}$ and then making a series of pseudocode modifications. One major modification was to substitute the code of the ideal key exchange oracles in wherever S made queries to them and to replace the tables $\vec{k}[]$ and $\vec{k}'[]$ with tables $pk_{\mathcal{I}}[]$, $pk_{\mathcal{R}}[]$, $sk[]$, $k_{\mathcal{R}}[]$, and $k'_{\mathcal{R}}[]$ that store each of their subcomponents. Then, the table $k_{\mathcal{R}}[]$ for non-shadow keys was eliminated. These keys were only used when returned by FING so their sampling was deferred until they were needed there (right above the boxed code of G_1).

$\underline{\textsc{Reg}(u)}$
$(\cdot,\cdot,\cdot,t) \leftarrow \sigma[u]$; $t \leftarrow t+1$
For $l = 1,\ldots,n$ do
  $(pk_{\mathcal{I}}, sk_{\mathcal{I}}) \leftarrow\!\!\$\ \mathsf{KE.Kg}_{\mathcal{I}}$
  $pk_{\mathcal{I}}[(u,t,l)] \leftarrow pk_{\mathcal{I}}$; $sk[(u,t,l)] \leftarrow sk_{\mathcal{I}}$
  $\mathbf{pk}[l] \leftarrow pk_{\mathcal{I}}$; $(\mathbf{ek}[l], \mathbf{dk}[l]) \leftarrow\!\!\$\ \mathsf{PKE.Kg}$
$\sigma[u] \leftarrow (\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t)$; $\mathbf{T}.\mathsf{add}(\textsc{Reg}, u, (\mathbf{pk}, \mathbf{ek}))$
Return $(\mathbf{pk}, \mathbf{ek})$

$\underline{\textsc{Req}(u,i,m,v)}$
$(\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t) \leftarrow \sigma[v]$; $(pk, ek, l) \leftarrow m$
If $(pk, ek) = (\mathbf{pk}[l], \mathbf{ek}[l])$
  $(pk_{\mathcal{R}}, sk_{\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{KE.Kg}_{\mathcal{R}}$; $k'_{\mathcal{R}} \leftarrow\!\!\$\ \{0,1\}^{\mathsf{KE.k}}$
  $(pk_{\mathcal{I}}[(u,i)], pk_{\mathcal{R}}[(u,i)]) \leftarrow (pk, pk_{\mathcal{R}})$
  $sk[(u,i)] \leftarrow sk_{\mathcal{R}}$; $k'_{\mathcal{R}}[(u,i)] \leftarrow k'_{\mathcal{R}}$
  $\sigma[u,i].\mathsf{lab} \leftarrow (u,i)$; $\sigma[u,i].\mathsf{pid} \leftarrow (v,t,l)$; $sk_{\mathcal{R}} \leftarrow \bot$
  $\vec{k} \leftarrow ((pk, pk_{\mathcal{R}}), sk_{\mathcal{R}}, k'_{\mathcal{R}})$; $c \leftarrow\!\!\$\ \mathsf{PKE.Enc}(ek, k'_{\mathcal{R}})$
Else
  $(pk_{\mathcal{R}}, sk_{\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{KE.Kg}_{\mathcal{R}}$; $k'_{\mathcal{R}} \leftarrow\!\!\$\ \{0,1\}^{\mathsf{KE.k}}$
  $\vec{k} \leftarrow ((pk, pk_{\mathcal{R}}), sk_{\mathcal{R}}, k'_{\mathcal{R}})$; $c \leftarrow\!\!\$\ \mathsf{PKE.Enc}(ek, k'_{\mathcal{R}})$
$z \leftarrow pk \,\|\, ek \,\|\, pk_{\mathcal{R}} \,\|\, c \,\|\, l$; $fp \leftarrow \mathsf{H}(z)$
$\sigma[u,i].\mathsf{keys} \leftarrow \vec{k}$; $\sigma[u,i].\mathsf{fing} \leftarrow z$
$\sigma[u, pk, ek, pk_{\mathcal{R}}, c, l] \leftarrow i$; $m' \leftarrow (pk_{\mathcal{R}}, c, l)$
$\mathbf{T}.\mathsf{add}(\textsc{Req}, (u,i,m,v), (m', fp))$; Return (m', fp)

$\underline{\textsc{Fin}(u,i,m,v)}$
$(pk, c, l) \leftarrow m$; $(\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t) \leftarrow \sigma[u]$; $fp \leftarrow \bot$
If $\mathbf{pk}[l] \neq \bot$ then
  $j \leftarrow \sigma[v, \mathbf{pk}[l], \mathbf{ek}[l], pk, c, l]$
  If $\sigma[v,j].\mathsf{lab} \neq \bot$
    $pk_{\mathcal{R}}[(u,t,l)] \leftarrow pk_{\mathcal{R}}[(v,j)]$
    $k'_{\mathcal{R}}[(u,t,l)] \leftarrow k'_{\mathcal{R}}[(v,j)]$
    $\sigma[u,i].\mathsf{lab} \leftarrow (u,t,l)$
    $\sigma[u,i].\mathsf{pid} \leftarrow (v,j)$; $sk_{\mathcal{I}} \leftarrow \bot$
  Else
    $\mathcal{X}.\mathsf{add}((u,t,l))$; $sk_{\mathcal{I}} \leftarrow sk[(u,t,l)]$
  $k_{\mathcal{R}} \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}[l], c)$
  $\vec{k} \leftarrow ((\mathbf{pk}[l], pk), sk_{\mathcal{I}}, k_{\mathcal{R}})$
  $z \leftarrow \mathbf{pk}[l] \,\|\, \mathbf{ek}[l] \,\|\, pk \,\|\, c \,\|\, l$; $fp \leftarrow \mathsf{H}(z)$
  $(\mathbf{pk}[l], \mathbf{ek}[l], \mathbf{dk}[l]) \leftarrow (\bot, \bot, \bot)$
  $\sigma[u] \leftarrow (\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t)$; $\sigma[u,i].\mathsf{keys} \leftarrow \vec{k}$; $\sigma[u,i].\mathsf{fing} \leftarrow z$
$\mathbf{T}.\mathsf{add}(\textsc{Fin}, (u,i,m,v), fp)$; Return fp

Figure 3.18. Oracles shared by G_0 and G_1.

$\underline{\text{FING}(u,i,v,j)}$

If $\sigma[u,i].\mathsf{fing} = \sigma[v,j].\mathsf{fing}$

  $y \leftarrow \mathsf{true}$; $\mathsf{lab}_1 \leftarrow \sigma[u,i].\mathsf{lab}$; $\mathsf{lab}_2 \leftarrow \sigma[v,j].\mathsf{lab}$

  If $\perp \in \{\mathsf{lab}_1, \mathsf{lab}_2\}$ then $y \leftarrow \mathsf{false}$

  If $\sigma[u,i].\mathsf{pid} \neq \mathsf{lab}_2$ or $\sigma[v,j].\mathsf{pid} \neq \mathsf{lab}_1$ then $y \leftarrow \mathsf{false}$

Else $(y, \mathsf{lab}_1, \mathsf{lab}_2) \leftarrow (\mathsf{false}, \perp, \perp)$

$t \leftarrow \mathsf{I}(\mathbf{T}, u, i, v, j)$; $t' \leftarrow []$

For $\mathsf{lab} \in \mathcal{X}$ do

  If $\mathsf{lab} = \mathsf{lab}_1$ then $t'.\mathsf{add}((u,i))$

  If $\mathsf{lab} = \mathsf{lab}_2$ then $t'.\mathsf{add}((v,j))$

If y and $t = t'$ then

  If $\{\mathsf{lab}_1, \mathsf{lab}_2\} \cap \mathcal{X} \neq \emptyset$ then $(k_1, k_2) \leftarrow (k'_{\mathcal{R}}[\mathsf{lab}_1], k'_{\mathcal{R}}[\mathsf{lab}_2])$

  Else

    $k_{\mathcal{R}} \leftarrow\$ \{0,1\}^{\mathsf{KE.k}}$; $(k_1, k_2) \leftarrow (k_{\mathcal{R}}, k_{\mathcal{R}})$

    $\boxed{(k_1, k_2) \leftarrow (k'_{\mathcal{R}}[\mathsf{lab}_1], k'_{\mathcal{R}}[\mathsf{lab}_2])}$

  $\vec{k}_1 \leftarrow ((pk_{\mathcal{I}}[\mathsf{lab}_1], pk_{\mathcal{R}}[\mathsf{lab}_1]), sk[\mathsf{lab}_1], k_1)$

  $\vec{k}_2 \leftarrow ((pk_{\mathcal{I}}[\mathsf{lab}_2], pk_{\mathcal{R}}[\mathsf{lab}_2]), sk[\mathsf{lab}_2], k_2)$

  $z \leftarrow (\vec{k}_1, \vec{k}_2)$

Else $z \leftarrow (\perp, \perp)$

$\mathbf{T}.\mathsf{add}(\text{FING}, (u,i,v,j), \varepsilon)$; Return z

$\underline{\text{CORRUPT}(u)}$

$(\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t) \leftarrow \sigma[u]$

For l s.t. $\mathbf{pk}[l] \neq \perp$ do $\mathcal{X}.\mathsf{add}((u,t,l))$; $\mathbf{sk}[l] \leftarrow sk[(u,t,l)]$

$st_u \leftarrow (\mathbf{pk}, \mathbf{sk}, \mathbf{ek}, \mathbf{dk})$

$\mathbf{T}.\mathsf{add}(\text{CORRUPT}, u, st_u)$

Return st_u

$\underline{\text{REVEAL}(u,i)}$

$((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk, k_{\mathcal{R}}) \leftarrow \sigma[u,i].\mathsf{keys}$

If $\sigma[u,i].\mathsf{lab} \neq \perp$ then

  $\mathcal{X}.\mathsf{add}(\sigma[u,i].\mathsf{lab})$; $sk \leftarrow sk[\sigma[u,i].\mathsf{lab}]$

  $\sigma[u,i].\mathsf{keys} \leftarrow ((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk, k_{\mathcal{R}})$

$\mathbf{T}.\mathsf{add}(\text{REVEAL}, (u,i), \sigma[u,i].\mathsf{keys})$

Return $\sigma[u,i].\mathsf{keys}$

Figure 3.19. Oracles shared by G_0 and G_1. Boxed code is only included in G_1.

The claim follows from observing that the pseudocode modifications do not modify the behavior of the game. The interested reader is referred to Section 3.9.2 where we walk through this in detail via three additional intermediate games.

```
P_mu^{New,Enc,Show,Chal,Key}
σ ←$ S(Init)
b' ←$ A^{RegS,ReqS,FinS,FingS,CorruptS,RevealS}
If V(T) then b' ← 0
Return b'

ReqS(u,i,m,v)
(pk, ek, dk, t) ← σ[v] ; (pk, ek, l) ← m
If (pk, ek) = (pk[l], ek[l])
   (pk_R, sk_R) ←$ KE.Kg_R
   (pk_I[(u,i)], pk_R[(u,i)]) ← (pk, pk_R)
   sk[(u,i)] ← sk_R
   σ[u,i].lab ← (u,i)
   σ[u,i].pid ← (v,t,l) ; sk_R ← ⊥
   k⃗ ← ((pk, pk_R), sk_R, ⋄)
   c ← Enc((v,t,l),(u,i))
Else
   (pk_R, sk_R) ←$ KE.Kg_R ; k'_R ←$ {0,1}^{KE.k}
   k⃗ ← ((pk, pk_R), sk_R, k'_R)
   c ←$ PKE.Enc(ek, k'_R)
z ← pk ‖ ek ‖ pk_R ‖ c ‖ l ; fp ← H(z)
σ[u,i].keys ← k⃗ ; σ[u,i].fing ← z
σ[u, pk, ek, pk_R, c, l] ← i ; m' ← (pk_R, c, l)
T.add(Req, (u,i,m,v), (m', fp))
Return (m', fp)

RegS(u)
(·,·,·,t) ← σ[u] ; t ← t+1
For l = 1,...,n do
   (pk_I, sk_I) ←$ KE.Kg_I
   pk_I[(u,t,l)] ← pk_I
   sk[(u,t,l)] ← sk_I ; pk[l] ← pk_I
   ek[l] ← New((u,t,l)) ; dk[l] ← ⋄
σ[u] ← (pk, ek, dk, t)
T.add(Reg, u, (pk, ek))
Return (pk, ek)

FinS(u,i,m,v)
(pk, c, l) ← m ; (pk, ek, dk, t) ← σ[u]
fp ← ⊥
If pk[l] ≠ ⊥ then
   j ← σ[v, pk[l], ek[l], pk, c, l]
   If σ[v,j].lab ≠ ⊥
      pk_R[(u,t,l)] ← pk_R[(v,j)]
      σ[u,i].lab ← (u,t,l)
      σ[u,i].pid ← (v,j) ; sk_I ← ⊥
   Else
      X.add((u,t,l)) ; sk_I ← sk[(u,t,l)]
      dk[l] ← Show((u,t,l))
   If dk[l] ≠ ⊥ then k_R ← PKE.Dec(dk[l], c)
   Else k_R ← ⋄ ; c[(u,i)] ← c
   k⃗ ← ((pk[l], pk), sk_I, k_R)
   z ← pk[l] ‖ ek[l] ‖ pk ‖ c ‖ l
   fp ← H(z)
   (pk[l], ek[l], dk[l]) ← (⊥, ⊥, ⊥)
   σ[u] ← (pk, ek, dk, t)
   σ[u,i].keys ← k⃗ ; σ[u,i].fing ← z
T.add(Fin, (u,i,m,v), fp) ; Return fp
```

Figure 3.20. Reduction to security of PKE.

Claim 2, $\Pr[G_1] - \Pr[G_0] \leq 2q \cdot \mathsf{Adv}^{\mathsf{ind\text{-}cpa}}_{\mathsf{PKE}}(\mathcal{P})$**.**

Game G_1 is also defined by the oracles in Figures 3.18 and 3.19. Note that it does include the boxed code. Thus games G_0 and G_1 differ only in the box code in FING. In G_0 fresh random keys are returned by unexposed sessions. In G_1 the shadows keys (which were previously encrypted in REQ) are returned instead. This difference corresponds naturally to the behavior of CHAL in $G^{\mathsf{mu\text{-}pke}}$ which returns a fresh random key if $b = 0$ and a previously encrypted key

$\underline{\text{FINGS}(u,i,v,j)}$
If $\sigma[u,i].\mathsf{fing} = \sigma[v,j].\mathsf{fing}$
  $y \leftarrow \mathsf{true}$; $\mathsf{lab}_1 \leftarrow \sigma[u,i].\mathsf{lab}$; $\mathsf{lab}_2 \leftarrow \sigma[v,j].\mathsf{lab}$
  If $\perp \in \{\mathsf{lab}_1, \mathsf{lab}_2\}$ then $y \leftarrow \mathsf{false}$
  If $\sigma[u,i].\mathsf{pid} \neq \mathsf{lab}_2$ or $\sigma[v,j].\mathsf{pid} \neq \mathsf{lab}_1$ then $y \leftarrow \mathsf{false}$
Else $(y, \mathsf{lab}_1, \mathsf{lab}_2) \leftarrow (\mathsf{false}, \perp, \perp)$
$t \leftarrow \mathsf{I}(\mathbf{T}, u, i, v, j)$; $t' \leftarrow []$
For $\mathsf{lab} \in \mathcal{X}$ do
  If $\mathsf{lab} = \mathsf{lab}_1$ then $t'.\mathsf{add}((u,i))$
  If $\mathsf{lab} = \mathsf{lab}_2$ then $t'.\mathsf{add}((v,j))$
If y and $t = t'$ then
  If $\{\mathsf{lab}_1, \mathsf{lab}_2\} \cap \mathcal{X} \neq \emptyset$
    If $|\mathsf{lab}_1| = 3$ then $k_{\mathcal{R}} \leftarrow \text{KEY}(\mathsf{lab}_1, \mathsf{lab}_2)$
    Else $k_{\mathcal{R}} \leftarrow \text{KEY}(\mathsf{lab}_2, \mathsf{lab}_1)$
  Else
    If $|\mathsf{lab}_1| = 3$ then $k_{\mathcal{R}} \leftarrow \text{CHAL}(\mathsf{lab}_1, \mathsf{lab}_2)$
    Else $k_{\mathcal{R}} \leftarrow \text{CHAL}(\mathsf{lab}_2, \mathsf{lab}_1)$
  $\vec{k}_1 \leftarrow ((pk_{\mathcal{I}}[\mathsf{lab}_1], pk_{\mathcal{R}}[\mathsf{lab}_1]), sk[\mathsf{lab}_1], k_{\mathcal{R}})$
  $\vec{k}_2 \leftarrow ((pk_{\mathcal{I}}[\mathsf{lab}_2], pk_{\mathcal{R}}[\mathsf{lab}_2]), sk[\mathsf{lab}_2], k_{\mathcal{R}})$
  $z \leftarrow (\vec{k}_1, \vec{k}_2)$
Else $z \leftarrow (\perp, \perp)$
$\mathbf{T}.\mathsf{add}(\text{FING}, (u,i,v,j), \varepsilon)$; Return z

$\underline{\text{CORRUPTS}(u)}$
$(\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t) \leftarrow \sigma[u]$
For l s.t. $\mathbf{pk}[l] \neq \perp$ do
  $\mathcal{X}.\mathsf{add}((u,t,l))$; $\mathbf{sk}[l] \leftarrow sk[(u,t,l)]$; $\mathbf{dk}[l] \leftarrow \text{SHOW}((u,t,l))$
$st_u \leftarrow (\mathbf{pk}, \mathbf{sk}, \mathbf{ek}, \mathbf{dk})$
$\mathbf{T}.\mathsf{add}(\text{CORRUPT}, u, st_u)$
Return st_u

$\underline{\text{REVEALS}(u,i)}$
$((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk, k_{\mathcal{R}}) \leftarrow \sigma[u,i].\mathsf{keys}$
If $\sigma[u,i].\mathsf{lab} \neq \perp$ then
  $\mathcal{X}.\mathsf{add}(\sigma[u,i].\mathsf{lab})$; $sk \leftarrow sk[\sigma[u,i].\mathsf{lab}]$
  If $k_{\mathcal{R}} = \diamond$ and $|\sigma[(u,i)].\mathsf{lab}| = 2$ then
    $k_{\mathcal{R}} \leftarrow \text{KEY}(\sigma[(u,i)].\mathsf{pid}, (u,i))$
  If $k_{\mathcal{R}} = \diamond$ and $|\sigma[(u,i)].\mathsf{lab}| \neq 2$ then
    $dk \leftarrow \text{SHOW}(\sigma[(u,i)].\mathsf{lab})$; $k_{\mathcal{R}} \leftarrow \mathsf{PKE.Dec}(dk, c[(u,i)])$
  $\sigma[u,i].\mathsf{keys} \leftarrow ((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk, k_{\mathcal{R}})$
$\mathbf{T}.\mathsf{add}(\text{REVEAL}, (u,i), \sigma[u,i].\mathsf{keys})$
Return $\sigma[u,i].\mathsf{keys}$

Figure 3.21. Additional oracles of reduction to security of PKE.

if $b = 1$.[4] As such we will reduce the difference between G_0 and G_1 to the advantage of an adversary $\mathcal{P}_{\mathsf{mu}}$ playing $\mathrm{G}^{\mathsf{mu\text{-}pke}}$. The claim then follows by applying Lemma 18 to the adversary.

Consider the adversary $\mathcal{P}_{\mathsf{mu}}$ shown in Figures 3.20 and 3.21. It runs $\mathcal{A}$ with simulated oracles. These oracles were created by making small modifications to replace code of G_0 and G_1 with calls to the oracles of $\mathcal{P}_{\mathsf{mu}}$. We use highlighting to indicate where these changes were made.

In REGS, instead of sampling new keys for PKE locally, $\mathcal{P}_{\mathsf{mu}}$ uses its NEW oracle to do so. Because it does not know the corresponding decryption key at this time, it instead stores $\diamond$ as a placeholder value.

In REQS the sampling and encryption of the shadow key $k'_{\mathcal{R}}$ is replaced with a query to ENC for the matching encryption key. Since $\mathcal{P}_{\mathsf{mu}}$ does not know $k'_{\mathcal{R}}$, it instead stores $\diamond$ as a placeholder. Note that these replacements are only done in the first branch of the if statement. In the second branch, $\sigma[u,i].\mathsf{lab}$ and $\sigma[u,i].\mathsf{pid}$ are not initialized so it is not possible to cause FING to return keys for this session. Thus $\mathcal{P}_{\mathsf{mu}}$ can simply generate $k'_{\mathcal{R}}$ locally in this case.

In FINS, if it is not possible to cause FING to return keys for this session SHOW is used to learn the corresponding decryption key. If the decryption key is known then $\mathcal{P}_{\mathsf{mu}}$ can decrypt the ciphertext to learn what key (u,i) just received. Otherwise it stores the ciphertext and uses $\diamond$ as a placeholder value for the key.

Simulating the other three oracles requires some care to make sure the placeholder $\diamond$ is replaced whenever the value it replaced would be returned to the adversary. This is done with KEY in FINGS and REVEALS and with SHOW in CORRUPTS and REVEALS. The various checks involving the length of lab_1 are used to make sure that the inputs to KEY or CHAL are in the same order as they were to ENC. The second case in REVEALS uses decryption to recover $k_{\mathcal{R}}$, because the key of that user was obtained by decrypting the ciphertext it received and is thus not necessarily equal to any keys that KEY would return.

If the labels were unexposed in FINGS then CHAL is used to produce the key that will be returned. This matches G_0 by returning a fresh random key if $b = 0$ and G_1 by returning a

[4]This natural correspondence is, of course, unsurprising since $\mathrm{G}^{\mathsf{mu\text{-}pke}}$ was defined specifically for this purpose.

previously encrypted key if $b = 1$.

We note that the various "Require" statements of $\mathcal{P}_{\mathsf{mu}}$'s oracles are never triggered. It never queries $\textsc{New}$ with the same input twice because t and l always change for a user u. It never queries $\textsc{Enc}$ twice with the same input because V^* prevents $\mathcal{A}$ from calling $\textsc{ReqS}$ twice with the same session and always calls it with an input for which the appropriate $\textsc{New}$ query was made because before it is called we have $\mathbf{pk}[l] \neq \bot$. It never calls $\textsc{Chal}$ and ($\textsc{Show}$ or $\textsc{Key}$) for contradicting inputs because the latter oracles are only queried when a session is either exposed or unable to make $\textsc{FingS}$ return keys. It never queries $\textsc{Show}$ without first making the appropriate $\textsc{New}$ query because $\textsc{Show}$ is only ever queries after a check that $\mathbf{pk}[l] \neq \bot$. It never queries $\textsc{Chal}$ without first making the appropriate $\textsc{Enc}$ query because keys will only ever be returned in $\textsc{FingS}$ for "matching" labels which requires one of them to have been created when the appropriate $\textsc{Enc}$ was made. It never queries $\textsc{Chal}$ for the same i twice because V^* prevents the same session from being queried to $\textsc{FingS}$ twice. It never queries $\textsc{Key}$ without first making the appropriate $\textsc{Enc}$ query because keys will only ever be returned in $\textsc{FingS}$ for "matching" labels which requires one of them to have been created when the appropriate $\textsc{Enc}$ was made and because the only way for $k_{\mathcal{R}}$ to equal $\diamond$ in $\textsc{RevealS}$ is if it was set to that in $\textsc{ReqS}$ or $\textsc{FinS}$ both of which could only occur if the appropriate $\textsc{Enc}$ query was made.

Consequently we have $\Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_0] = \mathsf{Adv}^{\mathsf{mu\text{-}pke}}_{\mathsf{PKE},\kappa}(\mathcal{P}_{\mathsf{mu}})$ where κ is the length of keys used by GKE. Claim (2) then follows by applying Lemma 18 to create $\mathcal{P}$, noting that $\mathcal{P}_{\mathsf{mu}}$ makes at most $q_{\textsc{Reg}}$ queries to its $\textsc{New}$ oracle.

Claim 3, $\Pr[\mathrm{G}_2] = \Pr[\mathrm{G}_1]$**.**

Game G_2 is defined by the oracles in Figures 3.22 and 3.23. Note that the boxed code is included in G_2. This game was created by making a number of changes to the pseudocode of G_1.

First we note that the shadow keys are the only keys used now, so we notationally simplify by getting rid of the prime and referring to them simply as the keys. In $\textsc{Req}$ the only code in the else branch is also run in the other branch so we move that code out of the conditional.

$\underline{\text{REG}(u)}$
$(\cdot,\cdot,\cdot,t) \leftarrow \sigma[u]$; $t \leftarrow t+1$
For $l = 1,\ldots,n$ do
  $(pk_{\mathcal{I}}, sk_{\mathcal{I}}) \leftarrow\$ \mathsf{KE.Kg}_{\mathcal{I}}$
  $pk_{\mathcal{I}}[(u,t,l)] \leftarrow pk_{\mathcal{I}}$; $sk[(u,t,l)] \leftarrow sk_{\mathcal{I}}$
  $\mathbf{pk}[l] \leftarrow pk_{\mathcal{I}}$; $(\mathbf{ek}[l], \mathbf{dk}[l]) \leftarrow\$ \mathsf{PKE.Kg}$
$\sigma[u] \leftarrow (\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t)$; $\mathbf{T}.\mathsf{add}(\text{REG}, u, (\mathbf{pk}, \mathbf{ek}))$
Return $(\mathbf{pk}, \mathbf{ek})$

$\underline{\text{REQ}(u,i,m,v)}$
$(\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t) \leftarrow \sigma[v]$; $(pk, ek, l) \leftarrow m$
$(pk_{\mathcal{R}}, sk_{\mathcal{R}}) \leftarrow\$ \mathsf{KE.Kg}_{\mathcal{R}}$; $k_{\mathcal{R}} \leftarrow\$ \{0,1\}^{\mathsf{KE.k}}$
$(pk_{\mathcal{I}}[(u,i)], pk_{\mathcal{R}}[(u,i)]) \leftarrow (pk, pk_{\mathcal{R}})$
$sk[(u,i)] \leftarrow sk_{\mathcal{R}}$; $k_{\mathcal{R}}[(u,i)] \leftarrow k_{\mathcal{R}}$
If $(pk, ek) = (\mathbf{pk}[l], \mathbf{ek}[l])$
  $\sigma[u,i].\mathsf{lab} \leftarrow (u,i)$; $\sigma[u,i].\mathsf{pid} \leftarrow (v,t,l)$
$\vec{k} \leftarrow ((pk, pk_{\mathcal{R}}), sk_{\mathcal{R}}, k_{\mathcal{R}})$; $c \leftarrow\$ \mathsf{PKE.Enc}(ek, k_{\mathcal{R}})$
$z \leftarrow pk \,\|\, ek \,\|\, pk_{\mathcal{R}} \,\|\, c \,\|\, l$; $fp \leftarrow \mathsf{H}(z)$
$\sigma[u,i].\mathsf{keys} \leftarrow \vec{k}$; $\sigma[u,i].\mathsf{fing} \leftarrow z$
$\sigma[u, pk, ek, pk_{\mathcal{R}}, c, l] \leftarrow i$; $m' \leftarrow (pk_{\mathcal{R}}, c, l)$
$\mathbf{T}.\mathsf{add}(\text{REQ}, (u,i,m,v), (m', fp))$; Return (m', fp)

$\underline{\text{FIN}(u,i,m,v)}$
$(pk, c, l) \leftarrow m$; $(\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t) \leftarrow \sigma[u]$; $fp \leftarrow \bot$
If $\mathbf{pk}[l] \neq \bot$ then
  $j \leftarrow \sigma[v, \mathbf{pk}[l], \mathbf{ek}[l], pk, c, l]$
  If $\sigma[v,j].\mathsf{lab} \neq \bot$
    $\sigma[u,i].\mathsf{lab} \leftarrow (u,t,l)$; $\sigma[u,i].\mathsf{pid} \leftarrow (v,j)$
  Else $\mathcal{X}.\mathsf{add}((u,t,l))$
  $pk_{\mathcal{R}}[(u,t,l)] \leftarrow pk$
  $k_{\mathcal{R}} \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}[l], c)$
  $k_{\mathcal{R}}[(u,t,l)] \leftarrow k_{\mathcal{R}}$
  $sk_{\mathcal{I}} \leftarrow sk[(u,t,l)]$
  $\vec{k} \leftarrow ((\mathbf{pk}[l], pk), sk_{\mathcal{I}}, k_{\mathcal{R}})$
  $z \leftarrow \mathbf{pk}[l] \,\|\, \mathbf{ek}[l] \,\|\, pk \,\|\, c \,\|\, l$; $fp \leftarrow \mathsf{H}(z)$
  $(\mathbf{pk}[l], \mathbf{ek}[l], \mathbf{dk}[l]) \leftarrow (\bot, \bot, \bot)$
  $\sigma[u] \leftarrow (\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t)$; $\sigma[u,i].\mathsf{keys} \leftarrow \vec{k}$; $\sigma[u,i].\mathsf{fing} \leftarrow z$
$\mathbf{T}.\mathsf{add}(\text{FIN}, (u,i,m,v), fp)$; Return fp

Figure 3.22. Oracles shared by G_2 and G_3.

We rewrite FIN so that $pk_{\mathcal{R}}[(v,j)]$ and $k'_{\mathcal{R}}[(v,j)]$ are no longer used to set $pk_{\mathcal{R}}[(u,t,l)]$ and $k'_{\mathcal{R}}[(u,t,l)]$. This assignment only occurs when $\sigma[v,j].\mathsf{lab} \neq \bot$ (and thus $j \neq \bot$) so the

```
FING(u,i,v,j)
If σ[u,i].fing = σ[v,j].fing
   y ← true ; lab_1 ← σ[u,i].lab ; lab_2 ← σ[v,j].lab
   If ⊥ ∈ {lab_1, lab_2} or σ[u,i].pid ≠ lab_2 or σ[v,j].pid ≠ lab_1
      bad_2 ← true ; [y ← false]
Else (y, lab_1, lab_2) ← (false, ⊥, ⊥)
t ← I(T, u, i, v, j) ; t' ← []
For lab ∈ X do
   If lab = lab_1 then t'.add((u,i))
   If lab = lab_2 then t'.add((v,j))
If y then
   z ← (σ[u,i].keys, σ[v,j].keys)
   If t ≠ t' then bad_1 ← true ; [z ← (⊥,⊥)]
Else z ← (⊥,⊥)
T.add(FING, (u,i,v,j), ε) ; Return z

CORRUPT(u)
(pk, ek, dk, t) ← σ[u]
For l s.t. pk[l] ≠ ⊥ do X.add((u,t,l)) ; sk[l] ← sk[(u,t,l)]
st_u ← (pk, sk, ek, dk)
T.add(CORRUPT, u, st_u)
Return st_u

REVEAL(u,i)
If σ[u,i].lab ≠ ⊥ then X.add(σ[u,i].lab)
T.add(REVEAL, (u,i), σ[u,i].keys)
Return σ[u,i].keys
```

Figure 3.23. Oracles shared by G_2 and G_3. Boxed code is only included in G_2.

session (v,j) must have been created in REQ. Then because $j = \sigma[v, \mathbf{pk}[l], \mathbf{ek}[l], pk, c, l]$, it must hold that $pk_{\mathcal{R}}[(v,j)] = \mathbf{pk}[l]$ and that c is an encryption of $k'_{\mathcal{R}}[(v,j)]$ under $\mathbf{ek}[l]$. So we can directly set $pk_{\mathcal{R}}[(u,t,l)]$ to $\mathbf{pk}[l]$ and $k'_{\mathcal{R}}[(u,t,l)]$ to the decryption of c using $\mathbf{dk}[l]$.

Then we note that $pk_{\mathcal{I}}[]$, $pk_{\mathcal{R}}[]$, and $k_{\mathcal{R}}[]$ are only ever accessed (in FING and FING) after checks requiring that the appropriate $\sigma[].\mathsf{lab}$ is not $\perp$. As such, we can arbitrarily set entries of these tables when $\sigma[].\mathsf{lab}$ is $\perp$. In particular, we use this justification to move their assignments out of the conditionals in REQ and FIN.[5] Similarly $sk[[]x]$ when x is a 2-tuple will only ever be accessed (in REVEAL and FING) after checks requiring $\sigma[].\mathsf{lab}$ is not $\perp$ so we can

[5]In FIN, when $j = \perp$ this just results in unnecessary statements storing $\perp$ in these tables.

move its assignment out of the conditional in REQ.

In REQ we get rid of the assignment $sk_{\mathcal{R}} \leftarrow \bot$. This is only ever used (via $\vec{k}$) to decide what is stored in $\sigma[u,i].\mathsf{keys}$ which, in turn, is only ever accessed in REVEAL. Therein, if $\sigma[u,i].\mathsf{lab} \neq \bot$ (which is the case whenever $sk_{\mathcal{R}} \leftarrow \bot$) this component of $\sigma[u,i].\mathsf{keys}$ is anyways overwritten. By a similar logic, we can always set $sk_{\mathcal{I}} \leftarrow sk[(u,t,l)]$ in FIN.

Having done so, we get eliminate the overwriting of part of $\sigma[u,i].\mathsf{keys}$ in REVEAL because it will already store the value it would be overwritten to. Additionally, we can rewrite FING to use $\sigma[].\mathsf{keys}$ instead of the tables $pk_{\mathcal{I}}[]$, $pk_{\mathcal{R}}[]$, $sk[]$, and $k_{\mathcal{R}}[]$. For a session created in REQ it is clear this has the correct values because it is set to store exactly the same values just stored in those tables. A similar observation holds in FIN only requiring noting additionally that $\mathbf{pk}[l]$ necessarily matches the value stored in $pk_{\mathcal{I}}[]$ during REG.

The final modification is that we can simplify FING and rewrite the logic to include the flags bad_1 and bad_2.

The claim follows from our arguments that the pseudocode modifications do not modify the behavior of the game.

Claim 4, $\Pr[G_3] - \Pr[G_2] \leq nq^2 \cdot 2^{-H_\infty(\mathsf{PKE})}$.

Game G_3 is again defined by the oracles in Figures 3.22 and 3.23, but it does not contain the boxed code. Note that games G_3 and G_2 are identical-until-bad. Thus by the fundamental lemma of game playing [16] we have $\Pr[G_3] - \Pr[G_2] \leq \Pr[G_2 \text{ sets } \mathsf{bad}]$. For ease of discussion in the pseudocode we write the bad flag as two separate flags, bad_1 and bad_2. We overload notation and let bad_x denote the event that G_2 sets bad_x and $\neg\mathsf{bad}$ denote its negation. We will use that $\Pr[\mathsf{bad}] = \Pr[\mathsf{bad}_2] + \Pr[\mathsf{bad}_1 \wedge \neg\mathsf{bad}_2]$.

We start by analyzing $\Pr[\mathsf{bad}_2]$. The flag bad_2 is set if two sessions have matching fingerprints and either undefined labels or labels that don't match each other's partner identifiers. The analysis of this requires considering a number of cases. In our arguments below we implicitly make use of the fact that V^* disallows fingerprint queries in which $(u,i) = (v,j)$ or in which

either session's fingerprint is $\perp$.

We will we show if bad_2 is set then one of four events (that we call $E_1, \ldots, E_4$). These events are that the same c was produced in two different $\textsc{Req}$ queries (E_1), the same $\mathbf{ek}[l]$ was sampled with the same l in two different $\textsc{Reg}$ queries (E_2), $\mathcal{A}$ predicted a c during a $\textsc{Fin}$ query that was later produced by a $\textsc{Req}$ query (E_3), or $\mathcal{A}$ predicted a $\mathbf{ek}[l]$ during a $\textsc{Req}$ query that was later produced by a $\textsc{Req}$ query (E_4). By using the union bound we can get the following bounds.

$$\Pr[E_1] \leq \binom{q_{\textsc{Req}}}{2} \cdot 2^{-\mathsf{H}_\infty(\mathsf{PKE.Enc})}$$

$$\Pr[E_2] \leq n \cdot \binom{q_{\textsc{Reg}}}{2} \cdot 2^{-\mathsf{H}_\infty(\mathsf{PKE.Kg})}$$

$$\Pr[E_3] \leq q_{\textsc{Fin}} \cdot q_{\textsc{Req}} \cdot 2^{-\mathsf{H}_\infty(\mathsf{PKE.Enc})}$$

$$\Pr[E_4] \leq q_{\textsc{Req}} \cdot q_{\textsc{Reg}} \cdot 2^{-\mathsf{H}_\infty(\mathsf{PKE.Kg})}$$

Then we have the following bound.

$$\begin{aligned}
\Pr[\mathsf{bad}_2] &\leq \Pr[E_1 \vee E_2 \vee E_3 \vee E_4] \\
&\leq \Pr[E_1] + \Pr[E_2] + \Pr[E_3] + \Pr[E_4] \\
&\leq nq^2 \cdot 2^{-\mathsf{H}_\infty(\mathsf{PKE})}
\end{aligned}$$

The last line follows from simple calculations.

First suppose that sessions (u,i) and (v,j) were both initiated in $\textsc{Req}$. Then for them to have the same fingerprint $\sigma[].\mathsf{fing}$ they must have both happened to create the same ciphertext c. Similarly suppose that sessions (u,i) and (v,j) were both initiated in $\textsc{Fin}$. Then for them to have the same fingerprints they must have both happened to sample the same encryption key ek during $\textsc{Req}$. From the entropy of $\mathsf{PKE.Kg}$ we can bound the probability of this by

Now we only need to worry about the case that one of the sessions, say $(u, sess)$, was created in FING and the other, (v, j), was created in REQ. Suppose (u, i) was created first. Then its fingerprint contains some ciphertext c provided for the adversary. If the fingerprint of $(v, j$ matches then it later produces the same ciphertext which $\mathcal{A}$ predicted. Thus we only need to concentrate on the case that (v, j) was created first.

Suppose $\sigma[v, j].\mathsf{lab} = \bot$. Then there must have been a $\text{REQ}(u)$ query before the $\text{FIN}(u, i, \cdot, v)$ query (otherwise the check $(pk, ek) = (\mathbf{pk}[l], \mathbf{ek}[l])$ would have held in FIN). This means that the ek provided by $\mathcal{A}$ was a prediction of an encryption key later sampled in REQ.

Suppose $\sigma[(v, j)].\mathsf{lab} \neq \bot$ and $\sigma[u, i].\mathsf{lab} = \bot$. Then in FIN it held that $\sigma[v, j'].\mathsf{lab} = \bot$. We use a prime to note that this value j' is not necessarily equal to j. In particular, if it was then this label could not have been $\bot$. So the only way for this to hold is if $\sigma[v, pk, ek, pk_{\mathcal{R}}, c, l]$ was overwritten since the time that j was stored in it. This requires a c collision to have occurred.

Now we can assume both labels are non-$\bot$ and only need to worry about whether they match the appropriate partner identifiers. We continue to use our earlier analysis to assume that (v, j) was created first.

Suppose $\sigma[u, i].\mathsf{lab} \neq \sigma[v, j].\mathsf{pid}$. Mirroring our analysis from just a moment ago the only way for this to occur is if the table entry $\sigma[v, pk, ek, pk_{\mathcal{R}}, c, l]$ checked in FIN was overwritten between when (v, j) and (u, i) were created which requires a collision in what ciphertexts were sampled in REQ.

Suppose $\sigma[v, j].\mathsf{lab} \neq \sigma[u, i].\mathsf{pid}$. In this case it must hold that $\sigma[v, j].\mathsf{lab} = (u, t, l)$ and $\sigma[u, i].\mathsf{pid} = (u, t', l)$ for some $t < t'$. Thus the same $ek[l]$ was resampled across different calls to REG. This completes our analysis of bad_2.

Now we analyze the flag bad_1 (in particular showing $\Pr[\mathsf{bad}_1 \wedge \neg \mathsf{bad}_2] = 0$). It is set if the sequences of exposures induced by the oracles queries of $\mathcal{A}$ (according to I) does not match the sequence of components added to $\mathcal{X}$ (in FIN, CORRUPT, and REVEAL) for two matching sessions (by which we mean two sessions with the same fingerprint and whose partner identifiers

equal each others non-$\perp$ labels).

Note that elements can be added to $\mathcal{X}$ in FIN, CORRUPT, and REVEAL while I^* will only add sessions to its output for the latter queries. However, (u,t,l) is only added to $\mathcal{X}$ in FIN when the corresponding label is not changed from $\perp$ setting bad_1 would only occur if bad_2 was already set so we only need to consider the other oracles.

Every query to REVEAL with input (u,i) or (v,j) would result in I^* adding the session to its output. Since $\neg\mathsf{bad}_2$ implies the sessions under consideration have non-$\perp$ labels, the analogous additions are performed by $\mathcal{X}$.

For CORRUPT, the corresponding session would be added to the output of I^* if "its role is not $\mathcal{R}$, the most recent $\mathrm{REG}(u)$ query is same as the $\mathrm{REG}(u)$ query preceding the creation of (u,i), and the session had not already been created at the time of the corruption." The first condition is matched by $\mathcal{X}$ because it only adds 3-tuples in CORRUPT and sessions with role $\mathcal{R}$ correspond to 2-tuples in $\mathcal{X}$. The second condition is matched by $\mathcal{X}$ because the value of t increments in each REQ query. The third condition is matched by $\mathcal{X}$ because $\mathbf{pk}[l]$ will is set to $\perp$ in the FIN query creating the corresponding session.

Claim 5, $\Pr[\mathrm{G}_3] - \Pr[\mathrm{G}_4] \leq \mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathcal{H})$**.**

Game G_4 is defined to use the same code as G_3 except for REQ and FIN which are redefined in Fig. 3.24. These oracle are identical to those of G_3 except that the highlighted code has replaced the code commented out on the line above. Consequently in G_4, the variables $\sigma[].\mathsf{fing}$ store the hash of what they did in G_3. Note that values of these variables are only referenced during the comparison at the beginning of FING. Thus these behavior of these games can only differ if different inputs to H are found which have the same hash. This admits a straightforward reduction to the collision resistance of H.

Let $\mathcal{H}$ behave as follows. It runs G_3. Then every time FING is called it checks if $\sigma[u,i].\mathsf{fing}$ and $\sigma[v,j].\mathsf{fing}$ are different, yet hash to the same value. If so it outputs these two strings. Clearly $\Pr[\mathrm{G}_3] - \Pr[\mathrm{G}_4] \leq \mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathcal{H})$.

```
REQ(u, i, m, v)
(pk, ek, dk, t) ← σ[v] ; (pk, ek, l) ← m
(pk_R, sk_R) ←$ KE.Kg_R ; k_R ←$ {0,1}^KE.k
(pk_I[(u,i)], pk_R[(u,i)]) ← (pk, pk_R)
sk[(u,i)] ← sk_R ; k_R[(u,i)] ← k_R
If (pk, ek) = (pk[l], ek[l])
  σ[u,i].lab ← (u,i) ; σ[u,i].pid ← (v,t,l)
k⃗ ← ((pk, pk_R), sk_R, k_R) ; c ←$ PKE.Enc(ek, k_R)
//z ← pk ‖ ek ‖ pk_R ‖ c ‖ l ; fp ← H(z)
z ← H(pk ‖ ek ‖ pk_R ‖ c ‖ l) ; fp ← z)
σ[u,i].keys ← k⃗ ; σ[u,i].fing ← z
σ[u, pk, ek, pk_R, c, l] ← i ; m' ← (pk_R, c, l)
T.add(REQ, (u,i,m,v), (m', fp)) ; Return (m', fp)

FIN(u, i, m, v)
(pk, c, l) ← m ; (pk, ek, dk, t) ← σ[u] ; fp ← ⊥
If pk[l] ≠ ⊥ then
  j ← σ[v, pk[l], ek[l], pk, c, l]
  If σ[v,j].lab ≠ ⊥
    σ[u,i].lab ← (u,t,l) ; σ[u,i].pid ← (v,j)
  Else X.add((u,t,l))
  pk_R[(u,t,l)] ← pk
  k_R ← PKE.Dec(dk[l], c)
  k_R[(u,t,l)] ← k_R
  sk_I ← sk[(u,t,l)]
  k⃗ ← ((pk[l], pk), sk_I, k_R)
  //z ← pk[l] ‖ ek[l] ‖ pk ‖ c ‖ l ; fp ← H(z)
  z ← H(pk[l] ‖ ek[l] ‖ pk ‖ c ‖ l) ; fp ← z
  (pk[l], ek[l], dk[l]) ← (⊥, ⊥, ⊥)
  σ[u] ← (pk, ek, dk, t) ; σ[u,i].keys ← k⃗ ; σ[u,i].fing ← z
T.add(FIN, (u,i,m,v), fp) ; Return fp
```

Figure 3.24. Oracles of G_4.

Claim 6, $\Pr[G^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},1}] = \Pr[G_4]$**.**

While perhaps not easy to verify, we claim that $G^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},1}$ and G_4 are equivalent. This can be verified by plugging the code of KE into G^{oke} and making various pseudocode changes to the obtained game and game G_4 until they are clearly equivalent. The details of this are somewhat tedious, so we refer the interested reader to Section 3.9.3 where we walk through this in detail.

3.9.2 More proof details (Claim 1)

Game H_0**.**

We start our analysis by considering the oracles shown in Figures 3.25, 3.26, and 3.27. The first two of these were obtained by naively plugging the code of S into game $\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},0}$. The latter was obtained by making two simplifications to the code of of the ideal key exchange oracles. First we remove the "Require" statements from the beginning of each oracle since S will never cause them to be triggered. Then we remove all references to the variable mst which was only ever referenced for use with the "Require" statements.

To prove that these modifications do not change the behavior of $\mathrm{G}^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},0}$ we need only justify our claim that the simulator never triggers the "Require" statements. Most of these require statements deal with the values of $mst[\cdot].\mathsf{status}$. The variable will always have values $\perp$, created, waiting, and accepted and can only hold those values in that order (with the possible exception of created if the label was first used as the second input to $\textsc{IReq}$).

Oracle $\textsc{IReg}$ is only ever queries in $\textsc{Reg}$ with tuples of the form (u,t,l). Because t is stored per user and incremented each time $\textsc{Reg}$ is queried, we can verify that the require statement of $\textsc{IReg}$ is never triggered.

Oracle $\textsc{IReq}$ is only ever queried in $\textsc{Req}$ with tuples of the form (v,t,l) and (u,i). The fact that $\mathbf{pk}[l] = pk \neq \perp$ held on the line before the call means that these table entries must have been created in $\textsc{Reg}$ and not yet erased in $\textsc{Fin}$, so $mst[(v,t,l)].\mathsf{status} = \mathsf{created}$. The restrictions of V^* require that this was the first oracle query to session (u,i), so $mst[(v,t,l)].\mathsf{status} = \perp$.

Oracle $\textsc{IFin}$ is only ever queries in $\textsc{Fin}$ with tuples of the form (u,t,l) and (v,j). Because $\mathbf{pk}[l] \neq \perp$ before the query we can verify $mst[(u,t,l)].\mathsf{status} = \mathsf{created}$. Because $\sigma[(v,j)] \neq \perp$ before the query, this must have been set in $\textsc{Req}$ after a query to $\textsc{IReq}$. This would have set $mst[(v,j)].\mathsf{status}$ to waiting, so it must now be either waiting or accepted.

Oracle $\textsc{Reveal}$ is queried in $\textsc{Fin}$, $\textsc{Corrupt}$, and $\textsc{Reveal}$. In $\textsc{Fin}$ and $\textsc{Corrupt}$ this query is only made when $\mathbf{pk}[l] \neq \perp$ holds so we can verify $mst[(u,t,l)].\mathsf{status} = \mathsf{created}$.

In REVEAL this query is only made when $\sigma[u,i].\mathsf{lab} \neq \bot$. The value is only ever set after a IREQ or IFIN query, so $mst[\sigma[u,i].\mathsf{lab}].\mathsf{status}$ was set to $\mathsf{waiting}$ and cannot be $\bot$. Furthermore $mst[\sigma[u,i].\mathsf{lab}].\mathsf{status}$ could only be set to $\mathsf{accepted}$ by IFING which is called by FING. However, V^* disallows queries to REVEAL after queries to FING so this is not possible.

Oracle IFING is queried jn FING. This query is only made when $\sigma[u,i].\mathsf{lab}$ and $\sigma[v,j].\mathsf{lab}$ are not $\bot$, thus both statuses must have been set to $\mathsf{waiting}$. Because V^* disallows multiple queries to FING for the same session they must still be $\mathsf{waiting}$ at the time of this query. That $mst[\mathsf{lab}_1].\mathsf{pid} = \mathsf{lab}_2$ and $mst[\mathsf{lab}_2].\mathsf{pid} = \mathsf{lab}_1$ hold follows from the fact that $\sigma[u,i].\mathsf{pid} = \mathsf{lab}_2$ and $\sigma[v,j].\mathsf{pid} = \mathsf{lab}_1$ held in FING. Here note that $\sigma[\cdot,\cdot].\mathsf{lab}$ is set after queries to IREQ or IFIN to exactly match how $mst[\cdot].\mathsf{pid}$ was set.

Game H_1**.**

The oracles of game H_1 are defined in Figures 3.28 and 3.29. This game was obtained by plugging the code of H_0's ideal key exchange in wherever they were called. Everywhere this occurred is indicated by highlighting in H_1.

Game H_2**.**

The oracles of game H_2 are defined in Figures 3.30 and 3.31. These oracles were obtained by introducing tables $pk_{\mathcal{I}}[]$, $pk_{\mathcal{R}}[]$, $sk[]$, $k_{\mathcal{R}}[]$, and $k'_{\mathcal{R}}[]$ to replace $\vec{k}[]$ and $\vec{k}'[]$. In particular $\vec{k}[]$ always stores tuples of the form $((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk, k_{\mathcal{R}})$ and $\vec{k}'[]$ always stores tuples of the form $((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk, k'_{\mathcal{R}})$ which only ever differs from $\vec{k}[]$ in the last value. So we have separates these variables out into five separate tables. Highlighting indicates everywhere this occurred.

We made one additional small change in REQ. The table entry $pk_{\mathcal{I}}[(u,i)]$ is set to pk instead of $pk_{\mathcal{I}}[v,t,l]$. To note that this does not change anything simply observer that pk equals $\mathbf{pk}[l]$ which will have been set the same value as $pk_{\mathcal{I}}[v,t,l]$ in the early query $\text{REG}(v)$.

Game G_0**.**

The oracles of game G_0 are defined in Figures 3.18 and 3.19. These oracles were obtained by making small changes to those of H_2. We indicate where these changes will occur using

$\underline{\textsc{Reg}(u)}$
$(\cdot,\cdot,\cdot,t) \leftarrow \sigma[u]$; $t \leftarrow t+1$
For $l = 1,\ldots,n$ do
  $\mathbf{pk}[l] \leftarrow\!\!\$\ \textsc{IReg}((u,t,l))$; $(\mathbf{ek}[l],\mathbf{dk}[l]) \leftarrow\!\!\$\ \mathsf{PKE.Kg}$
$\sigma[u] \leftarrow (\mathbf{pk},\mathbf{ek},\mathbf{dk},t)$; $\mathbf{T}.\mathsf{add}(\textsc{Reg},u,(\mathbf{pk},\mathbf{ek}))$
Return $(\mathbf{pk},\mathbf{ek})$

$\underline{\textsc{Req}(u,i,m,v)}$
$(\mathbf{pk},\mathbf{ek},\mathbf{dk},t) \leftarrow \sigma[v]$; $(pk,ek,l) \leftarrow m$
If $(pk,ek) = (\mathbf{pk}[l],\mathbf{ek}[l])$
  $(pk_{\mathcal{R}},k'_{\mathcal{R}}) \leftarrow \textsc{IReq}((v,t,l),(u,i))$
  $\sigma[u,i].\mathsf{lab} \leftarrow (u,i)$; $\sigma[u,i].\mathsf{pid} \leftarrow (v,t,l)$; $sk_{\mathcal{R}} \leftarrow \bot$
Else
  $(pk_{\mathcal{R}},sk_{\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{KE.Kg}_{\mathcal{R}}$; $k'_{\mathcal{R}} \leftarrow\!\!\$\ \{0,1\}^{\mathsf{KE.k}}$
$\vec{k} \leftarrow ((pk,pk_{\mathcal{R}}),sk_{\mathcal{R}},k'_{\mathcal{R}})$; $c \leftarrow\!\!\$\ \mathsf{PKE.Enc}(ek,k'_{\mathcal{R}})$
$z \leftarrow pk \,\|\, ek \,\|\, pk_{\mathcal{R}} \,\|\, c \,\|\, l$; $fp \leftarrow \mathsf{H}(z)$
$\sigma[u,i].\mathsf{keys} \leftarrow \vec{k}$; $\sigma[u,i].\mathsf{fing} \leftarrow z$
$\sigma[u,pk,ek,pk_{\mathcal{R}},c,l] \leftarrow i$
$m' \leftarrow (pk_{\mathcal{R}},c,l)$
$\mathbf{T}.\mathsf{add}(\textsc{Req},(u,i,m,v),(m',fp))$; Return (m',fp)

$\underline{\textsc{Fin}(u,i,m,v)}$
$(pk,c,l) \leftarrow m$; $(\mathbf{pk},\mathbf{ek},\mathbf{dk},t) \leftarrow \sigma[u]$; $fp \leftarrow \bot$
If $\mathbf{pk}[l] \neq \bot$ then
  $j \leftarrow \sigma[v,\mathbf{pk}[l],\mathbf{ek}[l],pk,c,l]$
  If $\sigma[v,j].\mathsf{lab} \neq \bot$
    $\textsc{IFin}((u,t,l),(v,j))$; $\sigma[u,i].\mathsf{lab} \leftarrow (u,t,l)$
    $\sigma[u,i].\mathsf{pid} \leftarrow (v,j)$; $sk_{\mathcal{I}} \leftarrow \bot$
  Else
    $(\cdot,sk_{\mathcal{I}},\cdot) \leftarrow \textsc{IReveal}((u,t,l))$
  $k_{\mathcal{R}} \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}[l],c)$
  $\vec{k} \leftarrow ((\mathbf{pk}[l],pk),sk_{\mathcal{I}},k_{\mathcal{R}})$
  $z \leftarrow \mathbf{pk}[l] \,\|\, \mathbf{ek}[l] \,\|\, pk \,\|\, c \,\|\, l$; $fp \leftarrow \mathsf{H}(z)$
  $(\mathbf{pk}[l],\mathbf{ek}[l],\mathbf{dk}[l]) \leftarrow (\bot,\bot,\bot)$
  $\sigma[u] \leftarrow (\mathbf{pk},\mathbf{ek},\mathbf{dk},t)$; $\sigma[u,i].\mathsf{keys} \leftarrow \vec{k}$; $\sigma[u,i].\mathsf{fing} \leftarrow z$
$\mathbf{T}.\mathsf{add}(\textsc{Fin},(u,i,m,v),fp)$; Return fp

Figure 3.25. Oracles of game H_0.

---

boxes in H_2.

First in $\textsc{Req}$ we will move the code setting $\vec{k}$ and c up to be included in each branch of the if statement separately. Clearly this does not change the behavior of the oracles.

Next we note that values of $k_{\mathcal{R}}[]$ are only ever affect the view of the adversary when

$\underline{\textsc{Fing}(u,i,v,j)}$
If $\sigma[u,i].\mathsf{fing} = \sigma[v,j].\mathsf{fing}$
  $y \leftarrow \mathsf{true}$; $\mathsf{lab}_1 \leftarrow \sigma[u,i].\mathsf{lab}$; $\mathsf{lab}_2 \leftarrow \sigma[v,j].\mathsf{lab}$
  If $\bot \in \{\mathsf{lab}_1, \mathsf{lab}_2\}$ then $y \leftarrow \mathsf{false}$
  If $\sigma[u,i].\mathsf{pid} \neq \mathsf{lab}_2$ or $\sigma[v,j].\mathsf{pid} \neq \mathsf{lab}_1$ then $y \leftarrow \mathsf{false}$
Else $(y, \mathsf{lab}_1, \mathsf{lab}_2) \leftarrow (\mathsf{false}, \bot, \bot)$
$t \leftarrow \mathsf{I}(\mathbf{T}, u, i, v, j)$; $t' \leftarrow []$
For $\mathsf{lab} \in \mathcal{X}$ do
  If $\mathsf{lab} = \mathsf{lab}_1$ then $t'.\mathsf{add}((u,i))$
  If $\mathsf{lab} = \mathsf{lab}_2$ then $t'.\mathsf{add}((v,j))$
If y and $t = t'$ then
  $(\vec{k}_{\mathcal{I}}, \vec{k}_{\mathcal{R}}) \leftarrow \mathrm{I}\textsc{Fing}(\mathsf{lab}_1, \mathsf{lab}_2)$; $z \leftarrow (\vec{k}_{\mathcal{I}}, \vec{k}_{\mathcal{R}})$
Else $z \leftarrow (\bot, \bot)$
$\mathbf{T}.\mathsf{add}(\textsc{Fing}, (u,i,v,j), \varepsilon)$; Return z

$\underline{\textsc{Corrupt}(u)}$
$(\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t) \leftarrow \sigma[u]$
For l s.t. $\mathbf{pk}[l] \neq \bot$ do $((\cdot,\cdot), \mathbf{sk}[l], \cdot) \leftarrow \mathrm{I}\textsc{Reveal}((u,t,l))$
$st_u \leftarrow (\mathbf{pk}, \mathbf{sk}, \mathbf{ek}, \mathbf{dk})$
$\mathbf{T}.\mathsf{add}(\textsc{Corrupt}, u, st_u)$
Return st_u

$\underline{\textsc{Reveal}(u,i)}$
$((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk, k_{\mathcal{R}}) \leftarrow \sigma[u,i].\mathsf{keys}$
If $\sigma[u,i].\mathsf{lab} \neq \bot$ then
  $((\cdot,\cdot), sk, \cdot) \leftarrow \mathrm{I}\textsc{Reveal}(\sigma[u,i].\mathsf{lab})$
  $\sigma[u,i].\mathsf{keys} \leftarrow ((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk, k_{\mathcal{R}})$
$\mathbf{T}.\mathsf{add}(\textsc{Reveal}, (u,i), \sigma[u,i].\mathsf{keys})$
Return $\sigma[u,i].\mathsf{keys}$

Figure 3.26. Oracles of game H_0.

---

being returned in FING. So in G_0 we defer sampling these values until they are needed in FING. The same value will be used for $k_{\mathcal{R}}[\mathsf{lab}_1]$ and $k_{\mathcal{R}}[\mathsf{lab}_2]$ there. This is the correct behavior which mirrors that of H_2. The oracle FING will only return keys when $\sigma[u,i].\mathsf{pid} = \sigma[v,j].\mathsf{lab}$ and $\sigma[v,j].\mathsf{pid} = \sigma[u,i].\mathsf{lab}$. This is only possible if one of the sessions was created by a FIN query which would have set $k_{\mathcal{R}}[\mathsf{lab}_1]$ and $k_{\mathcal{R}}[\mathsf{lab}_2]$ to have the same value.

$\underline{\text{IREG}(\mathsf{lab})}$
$(pk_{\mathcal{I}}, sk_{\mathcal{I}}) \leftarrow\!\!\$\, \mathsf{KE.Kg}_{\mathcal{I}}$; $\vec{k}[\mathsf{lab}] \leftarrow ((pk_{\mathcal{I}}, \bot), sk_{\mathcal{I}}, \bot)$
$\vec{k}'[\mathsf{lab}] \leftarrow ((pk_{\mathcal{I}}, \bot), sk_{\mathcal{I}}, \bot)$; Return $pk_{\mathcal{I}}$

$\underline{\text{IREQ}(\mathsf{lab}_1, \mathsf{lab}_2)}$
$((pk_{\mathcal{I}}, \cdot), \cdot, \cdot) \leftarrow \vec{k}[\mathsf{lab}_1]$; $(pk_{\mathcal{R}}, sk_{\mathcal{R}}) \leftarrow\!\!\$\, \mathsf{KE.Kg}_{\mathcal{R}}$
$k_{\mathcal{R}} \leftarrow\!\!\$\, \{0,1\}^{\mathsf{KE.k}}$; $k'_{\mathcal{R}} \leftarrow\!\!\$\, \{0,1\}^{\mathsf{KE.k}}$; $\ell \leftarrow (pk_{\mathcal{I}}, pk_{\mathcal{R}})$
$\vec{k}[\mathsf{lab}_2] \leftarrow (\ell, sk_{\mathcal{R}}, k_{\mathcal{R}})$; $\vec{k}'[\mathsf{lab}_2] \leftarrow (\ell, sk_{\mathcal{R}}, k'_{\mathcal{R}})$
Return $(pk_{\mathcal{R}}, k'_{\mathcal{R}})$

$\underline{\text{IFIN}(\mathsf{lab}_1, \mathsf{lab}_2)}$
$((pk_{\mathcal{I}}, \cdot), sk_{\mathcal{I}}, \cdot) \leftarrow \vec{k}[\mathsf{lab}_1]$; $((\cdot, pk_{\mathcal{R}}), \cdot, k_{\mathcal{R}}) \leftarrow \vec{k}[\mathsf{lab}_2]$
$((\cdot, \cdot), \cdot, k'_{\mathcal{R}}) \leftarrow \vec{k}'[\mathsf{lab}_2]$; $\vec{k}[\mathsf{lab}_1] \leftarrow ((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk_{\mathcal{I}}, k_{\mathcal{R}})$
$\vec{k}'[\mathsf{lab}_1] \leftarrow ((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk_{\mathcal{I}}, k'_{\mathcal{R}})$; Return ε

$\underline{\text{IREVEAL}(\mathsf{lab})}$
$\mathcal{X}.\mathsf{add}(\mathsf{lab})$; Return $\vec{k}'[\mathsf{lab}]$

$\underline{\text{IFING}(\mathsf{lab}_1, \mathsf{lab}_2)}$
If $\{\mathsf{lab}_1, \mathsf{lab}_2\} \cap \mathcal{X} \neq \emptyset$ then return $(\vec{k}'[\mathsf{lab}_1], \vec{k}'[\mathsf{lab}_2])$
Return $(\vec{k}[\mathsf{lab}_1], \vec{k}[\mathsf{lab}_2])$

Figure 3.27. Oracles of game H_0.

3.9.3 More proof details (Claim 6)

In this subsection we provide all of the details necessary to verify our claim that $\Pr[G^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},1}] = \Pr[G_4]$. To start, we have reprodued game G_4 (originally defined across Fig. 3.22, Fig. 3.23, and Fig. 3.24) in Fig. 3.32 and Fig. 3.33. We will iteratively make a number of pseudocode on this game. We use highlighting to indicate where the first changes will occur.

Game F_0**.**

The oracles of game F_0 are defined in Fig. 3.34 and Fig. 3.35. This game was obtained by making a number of simplifications to game G_4. First we consider FING. Note that the two if statements which set bad flags are now dead code. Working backwards from there, we can note that all references to labels, t, and t' are dead code. Thus we can see that FING in F_0 is just a simplified version of this oracle which removes dead code.

Once this is simplified, we note that values stored in $\mathcal{X}$ are no longer referenced so we

$\underline{\text{Reg}(u)}$

$(\cdot,\cdot,\cdot,t) \leftarrow \sigma[u]$; $t \leftarrow t+1$

For $l = 1,\ldots,n$ do

  $(pk_{\mathcal{I}}, sk_{\mathcal{I}}) \leftarrow\$ \mathsf{KE.Kg}_{\mathcal{I}}$; $\vec{k}[(u,t,l)] \leftarrow ((pk_{\mathcal{I}},\perp), sk_{\mathcal{I}},\perp)$

  $\vec{k}'[(u,t,l)] \leftarrow ((pk_{\mathcal{I}},\perp), sk_{\mathcal{I}},\perp)$; $\mathbf{pk}[l] \leftarrow pk_{\mathcal{I}}$

  $(\mathbf{ek}[l], \mathbf{dk}[l]) \leftarrow\$ \mathsf{PKE.Kg}$

$\sigma[u] \leftarrow (\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t)$; $\mathbf{T}.\mathsf{add}(\text{Reg}, u, (\mathbf{pk}, \mathbf{ek}))$

Return $(\mathbf{pk}, \mathbf{ek})$

$\underline{\text{Req}(u,i,m,v)}$

$(\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t) \leftarrow \sigma[v]$; $(pk, ek, l) \leftarrow m$

If $(pk, ek) = (\mathbf{pk}[l], \mathbf{ek}[l])$

  $((pk_{\mathcal{I}},\cdot),\cdot,\cdot) \leftarrow \vec{k}[(v,t,l)]$; $(pk_{\mathcal{R}}, sk_{\mathcal{R}}) \leftarrow\$ \mathsf{KE.Kg}_{\mathcal{R}}$

  $k_{\mathcal{R}} \leftarrow\$ \{0,1\}^{\mathsf{KE.k}}$; $k'_{\mathcal{R}} \leftarrow\$ \{0,1\}^{\mathsf{KE.k}}$; $\ell \leftarrow (pk_{\mathcal{I}}, pk_{\mathcal{R}})$

  $\vec{k}[(u,i)] \leftarrow (\ell, sk_{\mathcal{R}}, k_{\mathcal{R}})$; $\vec{k}'[(u,i)] \leftarrow (\ell, sk_{\mathcal{R}}, k'_{\mathcal{R}})$

  $\sigma[u,i].\mathsf{lab} \leftarrow (u,i)$; $\sigma[u,i].\mathsf{pid} \leftarrow (v,t,l)$; $sk_{\mathcal{R}} \leftarrow \perp$

Else

  $(pk_{\mathcal{R}}, sk_{\mathcal{R}}) \leftarrow\$ \mathsf{KE.Kg}_{\mathcal{R}}$; $k'_{\mathcal{R}} \leftarrow\$ \{0,1\}^{\mathsf{KE.k}}$

$\vec{k} \leftarrow ((pk, pk_{\mathcal{R}}), sk_{\mathcal{R}}, k'_{\mathcal{R}})$; $c \leftarrow\$ \mathsf{PKE.Enc}(ek, k'_{\mathcal{R}})$

$z \leftarrow pk \,\|\, ek \,\|\, pk_{\mathcal{R}} \,\|\, c \,\|\, l$; $fp \leftarrow \mathsf{H}(z)$

$\sigma[u,i].\mathsf{keys} \leftarrow \vec{k}$; $\sigma[u,i].\mathsf{fing} \leftarrow z$

$\sigma[u, pk, ek, pk_{\mathcal{R}}, c, l] \leftarrow i$; $m' \leftarrow (pk_{\mathcal{R}}, c, l)$

$\mathbf{T}.\mathsf{add}(\text{Req}, (u,i,m,v), (m', fp))$; Return (m', fp)

$\underline{\text{Fin}(u,i,m,v)}$

$(pk, c, l) \leftarrow m$; $(\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t) \leftarrow \sigma[u]$; $fp \leftarrow \perp$

If $\mathbf{pk}[l] \neq \perp$ then

  $j \leftarrow \sigma[v, \mathbf{pk}[l], \mathbf{ek}[l], pk, c, l]$

  If $\sigma[v,j].\mathsf{lab} \neq \perp$

    $((pk_{\mathcal{I}},\cdot), sk_{\mathcal{I}},\cdot) \leftarrow \vec{k}[(u,t,l)]$

    $((\cdot, pk_{\mathcal{R}}),\cdot, k_{\mathcal{R}}) \leftarrow \vec{k}[(v,j)]$

    $((\cdot,\cdot),\cdot, k'_{\mathcal{R}}) \leftarrow \vec{k}'[(v,j)]$

    $\vec{k}[(u,t,l)] \leftarrow ((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk_{\mathcal{I}}, k_{\mathcal{R}})$

    $\vec{k}'[(u,t,l)] \leftarrow ((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk_{\mathcal{I}}, k'_{\mathcal{R}})$

    $\sigma[u,i].\mathsf{lab} \leftarrow (u,t,l)$; $\sigma[u,i].\mathsf{pid} \leftarrow (v,j)$; $sk_{\mathcal{I}} \leftarrow \perp$

  Else

    $\mathcal{X}.\mathsf{add}((u,t,l))$; $(\cdot, sk_{\mathcal{I}},\cdot) \leftarrow \vec{k}'[(u,t,l)]$

  $k_{\mathcal{R}} \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}[l], c)$; $\vec{k} \leftarrow ((\mathbf{pk}[l], pk), sk_{\mathcal{I}}, k_{\mathcal{R}})$

  $z \leftarrow \mathbf{pk}[l] \,\|\, \mathbf{ek}[l] \,\|\, pk \,\|\, c \,\|\, l$; $fp \leftarrow \mathsf{H}(z)$

  $(\mathbf{pk}[l], \mathbf{ek}[l], \mathbf{dk}[l]) \leftarrow (\perp,\perp,\perp)$

  $\sigma[u] \leftarrow (\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t)$; $\sigma[u,i].\mathsf{keys} \leftarrow \vec{k}$; $\sigma[u,i].\mathsf{fing} \leftarrow z$

$\mathbf{T}.\mathsf{add}(\text{Fin}, (u,i,m,v), fp)$; Return fp

Figure 3.28. Oracles of game H_1. Highlighting indicates changes from H_0.

$\underline{\text{FING}(u,i,v,j)}$
If $\sigma[u,i].\mathsf{fing} = \sigma[v,j].\mathsf{fing}$
  $y \leftarrow \mathsf{true}$; $\mathsf{lab}_1 \leftarrow \sigma[u,i].\mathsf{lab}$; $\mathsf{lab}_2 \leftarrow \sigma[v,j].\mathsf{lab}$
  If $\perp \in \{\mathsf{lab}_1, \mathsf{lab}_2\}$ then $y \leftarrow \mathsf{false}$
  If $\sigma[u,i].\mathsf{pid} \neq \mathsf{lab}_2$ or $\sigma[v,j].\mathsf{pid} \neq \mathsf{lab}_1$ then $y \leftarrow \mathsf{false}$
Else $(y, \mathsf{lab}_1, \mathsf{lab}_2) \leftarrow (\mathsf{false}, \perp, \perp)$
$t \leftarrow \mathsf{I}(\mathbf{T}, u, i, v, j)$; $t' \leftarrow []$
For $\mathsf{lab} \in \mathcal{X}$ do
  If $\mathsf{lab} = \mathsf{lab}_1$ then $t'.\mathsf{add}((u,i))$
  If $\mathsf{lab} = \mathsf{lab}_2$ then $t'.\mathsf{add}((v,j))$
If y and $t = t'$ then
  If $\{\mathsf{lab}_1, \mathsf{lab}_2\} \cap \mathcal{X} \neq \emptyset$ then $z \leftarrow (\vec{k}'[\mathsf{lab}_1], \vec{k}'[\mathsf{lab}_2])$
  Else $z \leftarrow (\vec{k}[\mathsf{lab}_1], \vec{k}[\mathsf{lab}_2])$
Else $z \leftarrow (\perp, \perp)$
$\mathbf{T}.\mathsf{add}(\text{FING}, (u,i,v,j), \varepsilon)$; Return z

$\underline{\text{CORRUPT}(u)}$
$(\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t) \leftarrow \sigma[u]$
For l s.t. $\mathbf{pk}[l] \neq \perp$ do
  $\mathcal{X}.\mathsf{add}((u,t,l))$; $((\cdot,\cdot), \mathbf{sk}[l], \cdot) \leftarrow \vec{k}'[(u,t,l)]$
$st_u \leftarrow (\mathbf{pk}, \mathbf{sk}, \mathbf{ek}, \mathbf{dk})$
$\mathbf{T}.\mathsf{add}(\text{CORRUPT}, u, st_u)$
Return st_u

$\underline{\text{REVEAL}(u,i)}$
$((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk, k_{\mathcal{R}}) \leftarrow \sigma[u,i].\mathsf{keys}$
If $\sigma[u,i].\mathsf{lab} \neq \perp$ then
  $\mathcal{X}.\mathsf{add}(\sigma[u,i].\mathsf{lab})$; $((\cdot,\cdot), sk, \cdot) \leftarrow \vec{k}'[\sigma[u,i].\mathsf{lab}]$
  $\sigma[u,i].\mathsf{keys} \leftarrow ((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk, k_{\mathcal{R}})$
$\mathbf{T}.\mathsf{add}(\text{REVEAL}, (u,i), \sigma[u,i].\mathsf{keys})$
Return $\sigma[u,i].\mathsf{keys}$

Figure 3.29. Oracles of game H_1. Highlighting indicates changes from H_0.

can remove the code in FIN, CORRUPT, and REVEAL which modifies it. This gives us the REVEAL given in F_0.

Next we note that $\sigma[].\mathsf{pid}$ is assigned, but never referenced, so we can remove the assignments to it in REQ and FIN. Similarly $\sigma[].\mathsf{lab}$ when storing a 3-tuple (as assigned by FIN) is never reference because the only time that $\sigma[].\mathsf{lab}$ is referenced at all is in FIN where it could only be a 2-tuple or $\perp$. So we can remove the assignment to it in FIN. But this makes the if statement checking of $\sigma[].\mathsf{lab}$ in FIN dead code so we can remove it and the assignments to

```
Reg(u)
(·,·,·,t) ← σ[u] ; t ← t+1
For l = 1,...,n do
   (pk_I, sk_I) ←$ KE.Kg_I
   pk_I[(u,t,l)] ← pk_I ; sk[(u,t,l)] ← sk_I
   pk[l] ← pk_I ; (ek[l], dk[l]) ←$ PKE.Kg
σ[u] ← (pk, ek, dk, t) ; T.add(Reg, u, (pk, ek))
Return (pk, ek)

Req(u,i,m,v)
(pk, ek, dk, t) ← σ[v] ; (pk, ek, l) ← m
If (pk, ek) = (pk[l], ek[l])
   (pk_R, sk_R) ←$ KE.Kg_R ; [k_R ←$ {0,1}^KE.k] ; k'_R ←$ {0,1}^KE.k
   (pk_I[(u,i)], pk_R[(u,i)]) ← (pk, pk_R)
   sk[(u,i)] ← sk_R ; [k_R[(u,i)] ← k_R] ; k'_R[(u,i)] ← k'_R
   σ[u,i].lab ← (u,i) ; σ[u,i].pid ← (v,t,l) ; sk_R ← ⊥
Else
   (pk_R, sk_R) ←$ KE.Kg_R ; k'_R ←$ {0,1}^KE.k
[k⃗ ← ((pk, pk_R), sk_R, k'_R) ; c ←$ PKE.Enc(ek, k'_R)]
z ← pk ‖ ek ‖ pk_R ‖ c ‖ l ; fp ← H(z)
σ[u,i].keys ← k⃗ ; σ[u,i].fing ← z
σ[u, pk, ek, pk_R, c, l] ← i ; m' ← (pk_R, c, l)
T.add(Req, (u,i,m,v), (m', fp)) ; Return (m', fp)

Fin(u,i,m,v)
(pk, c, l) ← m ; (pk, ek, dk, t) ← σ[u] ; fp ← ⊥
If pk[l] ≠ ⊥ then
   j ← σ[v, pk[l], ek[l], pk, c, l]
   If σ[v,j].lab ≠ ⊥
      pk_R[(u,t,l)] ← pk_R[(v,j)]
      [k_R[(u,t,l)] ← k_R[(v,j)]]
      k'_R[(u,t,l)] ← k'_R[(v,j)]
      σ[u,i].lab ← (u,t,l) ; σ[u,i].pid ← (v,j) ; sk_I ← ⊥
   Else
      X.add((u,t,l)) ; sk_I ← sk[(u,t,l)]
   k_R ← PKE.Dec(dk[l], c)
   k⃗ ← ((pk[l], pk), sk_I, k_R)
   z ← pk[l] ‖ ek[l] ‖ pk ‖ c ‖ l ; fp ← H(z)
   (pk[l], ek[l], dk[l]) ← (⊥, ⊥, ⊥)
   σ[u] ← (pk, ek, dk, t) ; σ[u,i].keys ← k⃗ ; σ[u,i].fing ← z
T.add(Fin, (u,i,m,v), fp) ; Return fp
```

Figure 3.30. Oracles of game H_2. Highlighting indicates changes from H_1. Boxes indicated where code will change to become G_0.

$\underline{\text{FING}(u,i,v,j)}$
If $\sigma[u,i].\mathsf{fing} = \sigma[v,j].\mathsf{fing}$
  $y \leftarrow \mathsf{true}$; $\mathsf{lab}_1 \leftarrow \sigma[u,i].\mathsf{lab}$; $\mathsf{lab}_2 \leftarrow \sigma[v,j].\mathsf{lab}$
  If $\perp \in \{\mathsf{lab}_1, \mathsf{lab}_2\}$ then $y \leftarrow \mathsf{false}$
  If $\sigma[u,i].\mathsf{pid} \neq \mathsf{lab}_2$ or $\sigma[v,j].\mathsf{pid} \neq \mathsf{lab}_1$ then $y \leftarrow \mathsf{false}$
Else $(y, \mathsf{lab}_1, \mathsf{lab}_2) \leftarrow (\mathsf{false}, \perp, \perp)$
$t \leftarrow \mathsf{I}(\mathbf{T}, u, i, v, j)$; $t' \leftarrow []$
For $\mathsf{lab} \in \mathcal{X}$ do
  If $\mathsf{lab} = \mathsf{lab}_1$ then $t'.\mathsf{add}((u,i))$
  If $\mathsf{lab} = \mathsf{lab}_2$ then $t'.\mathsf{add}((v,j))$
If y and $t = t'$ then
  If $\{\mathsf{lab}_1, \mathsf{lab}_2\} \cap \mathcal{X} \neq \emptyset$ then
    $(k_1, k_2) \leftarrow (k'_{\mathcal{R}}[\mathsf{lab}_1], k'_{\mathcal{R}}[\mathsf{lab}_2])$
  Else $\boxed{(k_1, k_2) \leftarrow (k_{\mathcal{R}}[\mathsf{lab}_1], k_{\mathcal{R}}[\mathsf{lab}_2])}$
  $\vec{k}_1 \leftarrow ((pk_{\mathcal{I}}[\mathsf{lab}_1], pk_{\mathcal{R}}[\mathsf{lab}_1]), sk[\mathsf{lab}_1], k_1)$
  $\vec{k}_2 \leftarrow ((pk_{\mathcal{I}}[\mathsf{lab}_2], pk_{\mathcal{R}}[\mathsf{lab}_2]), sk[\mathsf{lab}_2], k_2)$
  $z \leftarrow (\vec{k}_1, \vec{k}_2)$
Else $z \leftarrow (\perp, \perp)$
$\mathbf{T}.\mathsf{add}(\text{FING}, (u,i,v,j), \varepsilon)$; Return z

$\underline{\text{CORRUPT}(u)}$
$(\mathbf{pk}, \mathbf{ek}, \mathbf{dk}, t) \leftarrow \sigma[u]$
For l s.t. $\mathbf{pk}[l] \neq \perp$ do $\mathcal{X}.\mathsf{add}((u,t,l))$; $\mathbf{sk}[l] \leftarrow sk[(u,t,l)]$
$st_u \leftarrow (\mathbf{pk}, \mathbf{sk}, \mathbf{ek}, \mathbf{dk})$
$\mathbf{T}.\mathsf{add}(\text{CORRUPT}, u, st_u)$
Return st_u

$\underline{\text{REVEAL}(u,i)}$
$((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk, k_{\mathcal{R}}) \leftarrow \sigma[u,i].\mathsf{keys}$
If $\sigma[u,i].\mathsf{lab} \neq \perp$ then
  $\mathcal{X}.\mathsf{add}(\sigma[u,i].\mathsf{lab})$; $sk \leftarrow sk[\sigma[u,i].\mathsf{lab}]$
  $\sigma[u,i].\mathsf{keys} \leftarrow ((pk_{\mathcal{I}}, pk_{\mathcal{R}}), sk, k_{\mathcal{R}})$
$\mathbf{T}.\mathsf{add}(\text{REVEAL}, (u,i), \sigma[u,i].\mathsf{keys})$
Return $\sigma[u,i].\mathsf{keys}$

Figure 3.31. Oracles of game H_2. Highlighting indicates changes from H_1. Boxes indicated where code will change to become G_0.

$\sigma[].\mathsf{lab}$ in REQ as well. This makes the assignment of j in FIN and an if statement in REQ dead code. Removing that if statement gets rid of the need to parse $\sigma[v]$ at the beginning of REQ. Removing the assignment to j in FIN removes the need for the table it is assigned from, so we can get rid of the assignment to that table in REQ.

$\underline{\text{REG}(u)}$
$(\cdot,\cdot,\cdot,t) \leftarrow \sigma[u]$; $t \leftarrow t+1$
For $l = 1,\ldots,n$ do
  $(pk_{\mathcal{I}}, sk_{\mathcal{I}}) \leftarrow\!\!\$\ \mathsf{KE.Kg}_{\mathcal{I}}$
  $pk_{\mathcal{I}}[(u,t,l)] \leftarrow pk_{\mathcal{I}}$; $sk[(u,t,l)] \leftarrow sk_{\mathcal{I}}$
  $\mathbf{pk}[l] \leftarrow pk_{\mathcal{I}}$; $(\mathbf{ek}[l],\mathbf{dk}[l]) \leftarrow\!\!\$\ \mathsf{PKE.Kg}$
$\sigma[u] \leftarrow (\mathbf{pk},\mathbf{ek},\mathbf{dk},t)$; $\mathbf{T}.\mathsf{add}(\text{REG}, u, (\mathbf{pk},\mathbf{ek}))$
Return $(\mathbf{pk},\mathbf{ek})$

$\underline{\text{REQ}(u,i,m,v)}$
$(\mathbf{pk},\mathbf{ek},\mathbf{dk},t) \leftarrow \sigma[v]$; $(pk,ek,l) \leftarrow m$
$(pk_{\mathcal{R}}, sk_{\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{KE.Kg}_{\mathcal{R}}$; $k_{\mathcal{R}} \leftarrow\!\!\$\ \{0,1\}^{\mathsf{KE.k}}$
$(pk_{\mathcal{I}}[(u,i)], pk_{\mathcal{R}}[(u,i)]) \leftarrow (pk, pk_{\mathcal{R}})$
$sk[(u,i)] \leftarrow sk_{\mathcal{R}}$; $k_{\mathcal{R}}[(u,i)] \leftarrow k_{\mathcal{R}}$
If $(pk,ek) = (\mathbf{pk}[l],\mathbf{ek}[l])$
  $\sigma[u,i].\mathsf{lab} \leftarrow (u,i)$; $\sigma[u,i].\mathsf{pid} \leftarrow (v,t,l)$
$\vec{k} \leftarrow ((pk,pk_{\mathcal{R}}), sk_{\mathcal{R}}, k_{\mathcal{R}})$; $c \leftarrow\!\!\$\ \mathsf{PKE.Enc}(ek,k_{\mathcal{R}})$
$z \leftarrow \mathsf{H}(pk \,\|\, ek \,\|\, pk_{\mathcal{R}} \,\|\, c \,\|\, l)$; $fp \leftarrow z)$
$\sigma[u,i].\mathsf{keys} \leftarrow \vec{k}$; $\sigma[u,i].\mathsf{fing} \leftarrow z$
$\sigma[u,pk,ek,pk_{\mathcal{R}},c,l] \leftarrow i$; $m' \leftarrow (pk_{\mathcal{R}},c,l)$
$\mathbf{T}.\mathsf{add}(\text{REQ},(u,i,m,v),(m',fp))$; Return (m',fp)

$\underline{\text{FIN}(u,i,m,v)}$
$(pk,c,l) \leftarrow m$; $(\mathbf{pk},\mathbf{ek},\mathbf{dk},t) \leftarrow \sigma[u]$; $fp \leftarrow \bot$
If $\mathbf{pk}[l] \neq \bot$ then
  $j \leftarrow \sigma[v,\mathbf{pk}[l],\mathbf{ek}[l],pk,c,l]$
  If $\sigma[v,j].\mathsf{lab} \neq \bot$
    $\sigma[u,i].\mathsf{lab} \leftarrow (u,t,l)$; $\sigma[u,i].\mathsf{pid} \leftarrow (v,j)$
  Else $\mathcal{X}.\mathsf{add}((u,t,l))$
  $pk_{\mathcal{R}}[(u,t,l)] \leftarrow pk$
  $k_{\mathcal{R}} \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}[l],c)$
  $k_{\mathcal{R}}[(u,t,l)] \leftarrow k_{\mathcal{R}}$
  $sk_{\mathcal{I}} \leftarrow sk[(u,t,l)]$
  $\vec{k} \leftarrow ((\mathbf{pk}[l],pk), sk_{\mathcal{I}}, k_{\mathcal{R}})$
  $z \leftarrow \mathsf{H}(\mathbf{pk}[l] \,\|\, \mathbf{ek}[l] \,\|\, pk \,\|\, c \,\|\, l)$; $fp \leftarrow z$
  $(\mathbf{pk}[l],\mathbf{ek}[l],\mathbf{dk}[l]) \leftarrow (\bot,\bot,\bot)$
  $\sigma[u] \leftarrow (\mathbf{pk},\mathbf{ek},\mathbf{dk},t)$; $\sigma[u,i].\mathsf{keys} \leftarrow \vec{k}$; $\sigma[u,i].\mathsf{fing} \leftarrow z$
$\mathbf{T}.\mathsf{add}(\text{FIN},(u,i,m,v),fp)$; Return fp

Figure 3.32. Reproduction of oracles of G_4. Highlighting indicates where code will change to become F_0.

Next note that $pk_{\mathcal{I}}[]$, $pk_{\mathcal{R}}[]$, and $k_{\mathcal{R}}[]$ are assigned to in REG, REQ, and FIN but never referenced. Thus we can get rid of these assignments. Similarly $sk[x]$ when x is a 2-tuple is

```
FING(u, i, v, j)
If σ[u,i].fing = σ[v,j].fing
   y ← true ; lab_1 ← σ[u,i].lab ; lab_2 ← σ[v,j].lab
   If ⊥ ∈ {lab_1, lab_2} or σ[u,i].pid ≠ lab_2 or σ[v,j].pid ≠ lab_1
      bad_2 ← true
Else (y, lab_1, lab_2) ← (false, ⊥, ⊥)
t ← I(T, u, i, v, j) ; t′ ← []
For lab ∈ X do
   If lab = lab_1 then t′.add((u,i))
   If lab = lab_2 then t′.add((v,j))
If y then
   z ← (σ[u,i].keys, σ[v,j].keys)
   If t ≠ t′ then bad_1 ← true
Else z ← (⊥, ⊥)
T.add(FING, (u,i,v,j), ε) ; Return z

CORRUPT(u)
(pk, ek, dk, t) ← σ[u]
For l s.t. pk[l] ≠ ⊥ do X.add((u,t,l)) ; sk[l] ← sk[(u,t,l)]
st_u ← (pk, sk, ek, dk)
T.add(CORRUPT, u, st_u)
Return st_u

REVEAL(u, i)
If σ[u,i].lab ≠ ⊥ then X.add(σ[u,i].lab)
T.add(REVEAL, (u,i), σ[u,i].keys)
Return σ[u,i].keys
```

Figure 3.33. Oracles of game G_4. Highlighting indicates where code will change to become F_0.

---

assigned in REQ, but never referenced so we can remove this.

We can simplify the ends of REQ and FIN to get rid of the intermediate value z being used to store fingerprints.

Finally we make some additions. Where these have occurred is indicated by boxes in F_0. We have added an additional table **sk** to the state $\sigma[u]$ for each u. Each $\mathbf{sk}[l]$ always stores the secret key corresponding to $\mathbf{pk}[l]$, but never has any effect on output provided to the adversary. To avoid a notational conflict, we have renamed the temporary **sk** in CORRUPT to $\mathbf{sk}'$.

$\underline{\text{REG}(u)}$
$\boxed{(\cdot,\cdot,\cdot,\cdot,t) \leftarrow \sigma[u]}$; $t \leftarrow t+1$
For $l = 1,\ldots,n$ do
  $(pk_{\mathcal{I}}, sk_{\mathcal{I}}) \leftarrow\$ \mathsf{KE.Kg}_{\mathcal{I}}$
  $\boxed{\mathbf{sk}[l] \leftarrow sk_{\mathcal{I}}}$
  $sk[(u,t,l)] \leftarrow sk_{\mathcal{I}}$
  $\mathbf{pk}[l] \leftarrow pk_{\mathcal{I}}$; $(\mathbf{ek}[l], \mathbf{dk}[l]) \leftarrow\$ \mathsf{PKE.Kg}$
$\boxed{\sigma[u] \leftarrow (\mathbf{pk}, \mathbf{sk}, \mathbf{ek}, \mathbf{dk}, t)}$; $\mathbf{T}.\mathsf{add}(\text{REG}, u, (\mathbf{pk}, \mathbf{ek}))$
Return $(\mathbf{pk}, \mathbf{ek})$

$\underline{\text{REQ}(u,i,m,v)}$
$(pk, ek, l) \leftarrow m$
$(pk_{\mathcal{R}}, sk_{\mathcal{R}}) \leftarrow\$ \mathsf{KE.Kg}_{\mathcal{R}}$; $k_{\mathcal{R}} \leftarrow\$ \{0,1\}^{\mathsf{KE.k}}$
$\vec{k} \leftarrow ((pk, pk_{\mathcal{R}}), sk_{\mathcal{R}}, k_{\mathcal{R}})$; $c \leftarrow\$ \mathsf{PKE.Enc}(ek, k_{\mathcal{R}})$
$fp \leftarrow \mathsf{H}(pk \,\|\, ek \,\|\, pk_{\mathcal{R}} \,\|\, c \,\|\, l)$
$\sigma[u,i].\mathsf{keys} \leftarrow \vec{k}$; $\sigma[u,i].\mathsf{fing} \leftarrow fp$
$m' \leftarrow (pk_{\mathcal{R}}, c, l)$
$\mathbf{T}.\mathsf{add}(\text{REQ}, (u,i,m,v), (m', fp))$; Return (m', fp)

$\underline{\text{FIN}(u,i,m,v)}$
$(pk, c, l) \leftarrow m$; $\boxed{(\mathbf{pk}, \mathbf{sk}, \mathbf{ek}, \mathbf{dk}, t) \leftarrow \sigma[u]}$; $fp \leftarrow \bot$
If $\mathbf{pk}[l] \neq \bot$ then
  $k_{\mathcal{R}} \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}[l], c)$
  $sk_{\mathcal{I}} \leftarrow sk[(u,t,l)]$
  $\vec{k} \leftarrow ((\mathbf{pk}[l], pk), sk_{\mathcal{I}}, k_{\mathcal{R}})$
  $fp \leftarrow \mathsf{H}(\mathbf{pk}[l] \,\|\, \mathbf{ek}[l] \,\|\, pk \,\|\, c \,\|\, l)$
  $\boxed{(\mathbf{pk}[l], \mathbf{sk}[l], \mathbf{ek}[l], \mathbf{dk}[l]) \leftarrow (\bot,\bot,\bot,\bot)}$
  $\boxed{\sigma[u] \leftarrow (\mathbf{pk}, \mathbf{sk}, \mathbf{ek}, \mathbf{dk}, t)}$
  $\sigma[u,i].\mathsf{keys} \leftarrow \vec{k}$; $\sigma[u,i].\mathsf{fing} \leftarrow fp$
$\mathbf{T}.\mathsf{add}(\text{FIN}, (u,i,m,v), fp)$; Return fp

Figure 3.34. Oracles of game F_0. Boxes indicates changes from G_4. Highlighting indicates where code will change to become F_1.

Game F_1**.**

The oracles of game F_1 are defined in Fig. 3.36 and Fig. 3.36. This game was obtained from F_0 by making two simplifications. First notice that in F_0, the entry $sk[(u,t,l)]$ is only ever accessed (in FIN and CORRUPT) with the value of t currently stored in $\sigma[u]$. Because of this, when it is accessed it is always equal to $\mathbf{sk}[l]$ as stored in the same $\sigma[u]$. So we get rid of all references to $sk[]$ and use $\mathbf{sk}[l]$ instead where needed.

```
FING(u,i,v,j)
If σ[u,i].fing = σ[v,j].fing then
  z ← (σ[u,i].keys, σ[v,j].keys)
Else z ← (⊥,⊥)
T.add(FING,(u,i,v,j),ε) ; Return z

CORRUPT(u)
[(pk,sk,ek,dk,t) ← σ[u]]
For l s.t. pk[l] ≠ ⊥ do sk'[l] ← sk[(u,t,l)]
st_u ← (pk,sk',ek,dk)
T.add(CORRUPT,u,st_u)
Return st_u

REVEAL(u,i)
T.add(REVEAL,(u,i),σ[u,i].keys)
Return σ[u,i].keys
```

Figure 3.35. Oracles of game F_0. Boxes indicates changes from G_4. Highlighting indicates where code will change to become F_1.

Having done this, t is no longer needed in $\sigma[u]$ so we stop using it. It appeared in REG, FIN, and CORRUPT when it was being read from or written into $\sigma[u]$. Additionally it was incremented in REG. Without t there is not longer a need for the initial parsing of $\sigma[u]$ in REG.

Games F_2 and F_3.

We consider our final two games together (and backwards). Game F_3 is shown in Fig. 3.38 and was obtained by plugging the code of KE into $G^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},1}$. We use highlighting to indicate everywhere this occurred. Oracles CORRUPT and REVEAL are omitted, but are unchanged from $G^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},1}$ and identical to those shown in Fig. 3.37 for F_2.

We describe the changes that were made to F_3 to create game F_2. We use We uses boxes in F_3 to show where these changes occur. In game F_3 the role, status, pid, and psess entries of sst are only ever written to, never read. So we can eliminate them from FIN, REQ, and FING. Additionally, some assignments in REQ and FIN were simplified or removed because they were either superfluous or just required some simple renaming of variables.

To finish this section we argue that games F_1 and F_2 equivalent. The primary observation towards this is that values stored in $\sigma[u]$ or $\sigma u,i$ (in F_1) can be exactly identified with values

```
REG(u)
For l = 1,...,n do
   (pk_I, sk_I) <-$ KE.Kg_I
   sk[l] <- sk_I
   pk[l] <- pk_I ; (ek[l], dk[l]) <-$ PKE.Kg
σ[u] <- (pk, sk, ek, dk)
T.add(REG, u, (pk, ek))
Return (pk, ek)

FIN(u, i, m, v)
(pk, c, l) <- m ; (pk, sk, ek, dk) <- σ[u]
fp <- ⊥
If pk[l] ≠ ⊥ then
   k_R <- PKE.Dec(dk[l], c)
   sk_I <- sk[l]
   k⃗ <- ((pk[l], pk), sk_I, k_R)
   fp <- H(pk[l] || ek[l] || pk || c || l)
   (pk[l], sk[l], ek[l], dk[l]) <- (⊥, ⊥, ⊥, ⊥)
   σ[u] <- (pk, sk, ek, dk)
   σ[u,i].keys <- k⃗ ; σ[u,i].fing <- fp
T.add(FIN, (u, i, m, v), fp) ; Return fp

REQ(u, i, m, v)
(pk, ek, l) <- m
(pk_R, sk_R) <-$ KE.Kg_R ; k_R <-$ {0,1}^KE.k
k⃗ <- ((pk, pk_R), sk_R, k_R)
c <-$ PKE.Enc(ek, k_R)
fp <- H(pk || ek || pk_R || c || l)
σ[u,i].keys <- k⃗ ; σ[u,i].fing <- fp
m' <- (pk_R, c, l)
T.add(REQ, (u, i, m, v), (m', fp))
Return (m', fp)

FING(u, i, v, j)
If σ[u,i].fing = σ[v,j].fing then
   z <- (σ[u,i].keys, σ[v,j].keys)
Else z <- (⊥, ⊥)
T.add(FING, (u, i, v, j), ε) ; Return z

CORRUPT(u)
(pk, sk, ek, dk) <- σ[u]
For l s.t. pk[l] ≠ ⊥ do sk'[l] <- sk[l]
st_u <- (pk, sk', ek, dk)
T.add(CORRUPT, u, st_u)
Return st_u

REVEAL(u, i)
T.add(REVEAL, (u, i), σ[u,i].keys)
Return σ[u,i].keys
```

Figure 3.36. Oracles of game F_1. Highlighting indicates where code will change to become F_1.

stored in st_u of sst_u^i (in F_2). We have used highlighting in F_1 to indicate where these variable occur. It may seem like there are some differences in CORRUPT, but note that the temporary table $\mathbf{sk}'$ in F_1 is in fact being set to be exactly equivalent to the $\mathbf{sk}$ stored in $\sigma[u]$.

One this is identified the oracles REG, REQ, FING, and REVEAL in F_2 are obtained by simple rewriting of the corresponding oracles in F_1. Oracle FIN similarly requires simple rewriting. The primary difference is that the setting of $\sigma[(u,i)]$.keys and $\sigma[(u,i)]$.fing are implicit in F_1 because they are never explicitly set. In F_2 the corresponding sst_u^i variables are explicitly set to $\perp$. Additionally there was some unnoteworthy changing of names of temporary variables between the two. Oracle CORRUPT is again simple rewriting by making use of our observation

$\underline{\text{Reg}(u)}$
For $l = 1, \ldots, n$ do
  $(\mathbf{pk}[l], \mathbf{sk}[l]) \leftarrow\!\$\ \mathsf{KE.Kg}_{\mathcal{I}}$
  $(\mathbf{ek}[l], \mathbf{dk}[l]) \leftarrow\!\$\ \mathsf{PKE.Kg}$
$(st_u, m) \leftarrow ((\mathbf{pk}, \mathbf{sk}, \mathbf{ek}, \mathbf{dk}), (\mathbf{pk}, \mathbf{ek}))$
$\mathbf{T}.\mathsf{add}(\text{Reg}, u, m)$
Return m

$\underline{\text{Req}(u, i, m, v)}$
$(pk, ek, l) \leftarrow m$
$(pk_{\mathcal{R}}, sk_{\mathcal{R}}) \leftarrow\!\$\ \mathsf{KE.Kg}_{\mathcal{R}}$
$k_{\mathcal{R}} \leftarrow\!\$\ \{0,1\}^{\mathsf{KE.k}}$
$\vec{k} \leftarrow ((pk, pk_{\mathcal{R}}), sk_{\mathcal{R}}, k_{\mathcal{R}})$
$c \leftarrow\!\$\ \mathsf{PKE.Enc}(ek, k_{\mathcal{R}})$
$fp \leftarrow \mathsf{H}(pk \,\|\, ek \,\|\, pk_{\mathcal{R}} \,\|\, c \,\|\, l)$
$m' \leftarrow (pk_{\mathcal{R}}, c, l)$
$sst_u^i.\mathsf{keys} \leftarrow \vec{k}$
$sst_u^i.\mathsf{fing} \leftarrow fp$
$\mathbf{T}.\mathsf{add}(\text{Req}, (u, i, m, v), (m', fp))$
Return (m', fp)

$\underline{\text{Corrupt}(u)}$
$\mathbf{T}.\mathsf{add}(\text{Corrupt}, u, st_u)$
Return st_u

$\underline{\text{Reveal}(u, i)}$
$\mathbf{T}.\mathsf{add}(\text{Reveal}, (u, i), sst_u^i.\mathsf{keys})$
Return $sst_u^i.\mathsf{keys}$

$\underline{\text{Fin}(u, i, m, v)}$
$(pk_{\mathcal{R}}, c, l) \leftarrow m$
$(\mathbf{pk}, \mathbf{sk}, \mathbf{ek}, \mathbf{dk}) \leftarrow st_u$
If $\mathbf{pk}[l] = \bot$ then
  $(\vec{k}, fp) \leftarrow (\bot, \bot)$
Else
  $k_{\mathcal{R}} \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}[l], c)$
  $\vec{k} \leftarrow ((\mathbf{pk}[l], pk_{\mathcal{R}}), \mathbf{sk}[l], k_{\mathcal{R}})$
  $(pk, ek) \leftarrow (\mathbf{pk}[l], \mathbf{ek}[l])$
  $fp \leftarrow \mathsf{H}(pk \,\|\, ek \,\|\, pk_{\mathcal{R}} \,\|\, c \,\|\, l)$
  $(\mathbf{pk}[l], \mathbf{sk}[l], \mathbf{ek}[l], \mathbf{dk}[l]) \leftarrow (\bot, \bot, \bot, \bot)$
  $st_u \leftarrow (\mathbf{pk}, \mathbf{sk}, \mathbf{ek}, \mathbf{dk})$
$sst_u^i.\mathsf{keys} \leftarrow \vec{k}$
$sst_u^i.\mathsf{fing} \leftarrow fp$
$\mathbf{T}.\mathsf{add}(\text{Fin}, (u, i, m, v), fp)$
Return fp

$\underline{\text{Fing}(u, i, v, j)}$
If $sst_u^i.\mathsf{fing} = sst_v^j.\mathsf{fing}$ then
  $z \leftarrow (sst_u^i.\mathsf{keys}, sst_v^j.\mathsf{keys})$
Else
  $z \leftarrow (\bot, \bot)$
$\mathbf{T}.\mathsf{add}(\text{Fing}, (u, i, v, j), \varepsilon)$
Return z

Figure 3.37. Oracles of game F_2.

that the temporary $\mathbf{sk}'$ in F_1 is being set equal to the already existing $\mathbf{sk}$.

$\underline{\text{REQ}(u,i,m,v)}$
$(pk, ek, l) \leftarrow m$
$(pk_{\mathcal{R}}, sk_{\mathcal{R}}) \leftarrow\!\$\; \mathsf{KE.Kg}_{\mathcal{R}}$
$k_{\mathcal{R}} \leftarrow\!\$\; \{0,1\}^{\mathsf{KE.k}}$
$\vec{k} \leftarrow ((pk, pk_{\mathcal{R}}), sk_{\mathcal{R}}, k_{\mathcal{R}})$
$c \leftarrow\!\$\; \mathsf{PKE.Enc}(ek, k_{\mathcal{R}})$
$fp \leftarrow \mathsf{H}(pk \,\|\, ek \,\|\, pk_{\mathcal{R}} \,\|\, c \,\|\, l)$
$\boxed{(st_u, m', \vec{k}_u, fp) \leftarrow (st_u, (pk_{\mathcal{R}}, c, l), \vec{k}, fp)}$
$sst_u^i.\mathsf{keys} \leftarrow \vec{k}_u$
$sst_u^i.\mathsf{fing} \leftarrow fp$
$\boxed{sst_u^i.\mathsf{role} \leftarrow \mathcal{R}}$
$\boxed{sst_u^i.\mathsf{pid} \leftarrow v}$
If $\vec{k}_u = \bot$ then
  $\boxed{sst_u^i.\mathsf{status} \leftarrow \mathsf{rejected}}$
Else
  $\boxed{sst_u^i.\mathsf{status} \leftarrow \mathsf{waiting}}$
$\mathbf{T}.\mathsf{add}(\text{REQ}, (u,i,m,v), (m', fp))$
Return (m', fp)

$\underline{\text{FING}(u,i,v,j)}$
If $sst_u^i.\mathsf{fing} = sst_v^j.\mathsf{fing}$ then
  $\boxed{sst_u^i.\mathsf{status} \leftarrow \mathsf{accepted}}$
  $\boxed{sst_v^j.\mathsf{status} \leftarrow \mathsf{accepted}}$
  $\boxed{sst_u^i.\mathsf{psess} \leftarrow j \,;\, sst_v^j.\mathsf{psess} \leftarrow i}$
  $z \leftarrow (sst_u^i.\mathsf{keys}, sst_v^j.\mathsf{keys})$
Else
  $\boxed{sst_u^i.\mathsf{status} \leftarrow \mathsf{rejected}}$
  $\boxed{sst_v^j.\mathsf{status} \leftarrow \mathsf{rejected}}$
  $z \leftarrow (\bot, \bot)$
$\mathbf{T}.\mathsf{add}(\text{FING}, (u,i,v,j), \varepsilon)$
Return z

$\underline{\text{REG}(u)}$
For $l = 1, \ldots, n$ do
  $(\mathbf{pk}[l], \mathbf{sk}[l]) \leftarrow\!\$\; \mathsf{KE.Kg}_{\mathcal{I}}$
  $(\mathbf{ek}[l], \mathbf{dk}[l]) \leftarrow\!\$\; \mathsf{PKE.Kg}$
$(st_u, m) \leftarrow ((\mathbf{pk}, \mathbf{sk}, \mathbf{ek}, \mathbf{dk}), (\mathbf{pk}, \mathbf{ek}))$
$\mathbf{T}.\mathsf{add}(\text{REG}, u, m)$
Return m

$\underline{\text{FIN}(u,i,m,v)}$
$\boxed{st \leftarrow st_u}$
$(pk_{\mathcal{R}}, c, l) \leftarrow m$
$(\mathbf{pk}, \mathbf{sk}, \mathbf{ek}, \mathbf{dk}) \leftarrow st$
If $\mathbf{pk}[l] = \bot$ then
  $\boxed{(st_u, \vec{k}_u, fp) \leftarrow (st, \bot, \bot)}$
Else
  $k_{\mathcal{R}} \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}[l], c)$
  $\vec{k} \leftarrow ((\mathbf{pk}[l], pk_{\mathcal{R}}), \mathbf{sk}[l], k_{\mathcal{R}})$
  $(pk, ek) \leftarrow (\mathbf{pk}[l], \mathbf{ek}[l])$
  $fp \leftarrow \mathsf{H}(pk \,\|\, ek \,\|\, pk_{\mathcal{R}} \,\|\, c \,\|\, l)$
  $(\mathbf{pk}[l], \mathbf{sk}[l], \mathbf{ek}[l], \mathbf{dk}[l]) \leftarrow (\bot, \bot, \bot, \bot)$
  $st \leftarrow (\mathbf{pk}, \mathbf{sk}, \mathbf{ek}, \mathbf{dk})$
  $\boxed{(st_u, \vec{k}_u, fp) \leftarrow (st, \vec{k}, fp)}$
$sst_u^i.\mathsf{keys} \leftarrow \vec{k}_u$
$sst_u^i.\mathsf{fing} \leftarrow fp$
$\boxed{sst_u^i.\mathsf{role} \leftarrow \mathcal{I}}$
$\boxed{sst_u^i.\mathsf{pid} \leftarrow v}$
If $\vec{k}_u = \bot$ then
  $\boxed{sst_u^i.\mathsf{status} \leftarrow \mathsf{rejected}}$
Else
  $\boxed{sst_u^i.\mathsf{status} \leftarrow \mathsf{waiting}}$
$\mathbf{T}.\mathsf{add}(\text{FIN}, (u,i,m,v), fp)$
Return fp

Figure 3.38. Oracles of game F_3. Highlighting indicates where KE was plugged into $G^{\mathsf{oke}}_{\mathsf{KE},\mathcal{P},\mathsf{S},1}$. Boxes indicate differences from F_2.

3.10 Acknowledgements

Chapter 3, in full, has been submitted for publication of the material as it may appear as "Key Exchange for Messaging Apps," Jaeger, Joseph. The dissertation author was the primary investigator and author of this paper. The dissertation author was supported in part by NSF grants

CNS-1717640 and CNS-1526801.

Bibliography

[1] Michel Abdalla, Mihir Bellare, and Phillip Rogaway. The oracle Diffie-Hellman assumptions and an analysis of DHIES. In David Naccache, editor, *CT-RSA 2001*, volume 2020 of *LNCS*, pages 143–158, San Francisco, CA, USA, April 8–12, 2001. Springer, Heidelberg, Germany. Full version at http://web.cs.ucdavis.edu/~rogaway/papers/dhies.pdf.

[2] 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, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany.

[3] Donald Beaver and Stuart Haber. Cryptographic protocols provably secure against dynamic adversaries. In Rainer A. Rueppel, editor, *EUROCRYPT'92*, volume 658 of *LNCS*, pages 307–323, Balatonfüred, Hungary, May 24–28, 1993. Springer, Heidelberg, Germany.

[4] Mihir Bellare, Alexandra Boldyreva, and Silvio Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 259–274, Bruges, Belgium, May 14–18, 2000. Springer, Heidelberg, Germany.

[5] Mihir Bellare, Anand Desai, Eric Jokipii, and Phillip Rogaway. A concrete security treatment of symmetric encryption. In *38th FOCS*, pages 394–403, Miami Beach, Florida, October 19–22, 1997. IEEE Computer Society Press.

[6] Mihir Bellare, Anand Desai, David Pointcheval, and Phillip Rogaway. Relations among notions of security for public-key encryption schemes. In Hugo Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 26–45, Santa Barbara, CA, USA, August 23–27, 1998. Springer, Heidelberg, Germany.

[7] Mihir Bellare, Shanshan Duan, and Adriana Palacio. Key insulation and intrusion resilience over a public channel. In Marc Fischlin, editor, *CT-RSA 2009*, volume 5473 of *LNCS*, pages 84–99, San Francisco, CA, USA, April 20–24, 2009. Springer, Heidelberg, Germany.

[8] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. Breaking and provably repairing the ssh authenticated encryption scheme: A case study of the encode-then-encrypt-and-mac paradigm. *ACM Transactions on Information and System Security (TISSEC)*, 7(2):206–241, 2004.

[9] Mihir Bellare and Sara K. Miner. A forward-secure digital signature scheme. In Michael J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 431–448, Santa Barbara, CA, USA, August 15–19, 1999. Springer, Heidelberg, Germany.

[10] Mihir Bellare, Adam O'Neill, and Igors Stepanovs. Forward-security under continual leakage. Cryptology ePrint Archive, Report 2017/476, 2017. http://eprint.iacr.org/2017/476.

[11] Mihir Bellare, Kenneth G. Paterson, and Phillip Rogaway. Security of symmetric encryption against mass surveillance. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 1–19, Santa Barbara, CA, USA, August 17–21, 2014. Springer, Heidelberg, Germany.

[12] Mihir Bellare, David Pointcheval, and Phillip Rogaway. Authenticated key exchange secure against dictionary attacks. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 139–155, Bruges, Belgium, May 14–18, 2000. Springer, Heidelberg, Germany.

[13] Mihir Bellare, Thomas Ristenpart, and Stefano Tessaro. Multi-instance security and its application to password-based cryptography. In Reihaneh Safavi-Naini and Ran Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 312–329, Santa Barbara, CA, USA, August 19–23, 2012. Springer, Heidelberg, Germany.

[14] Mihir Bellare and Phillip Rogaway. Random oracles are practical: A paradigm for designing efficient protocols. In Dorothy E. Denning, Raymond Pyle, Ravi Ganesan, Ravi S. Sandhu, and Victoria Ashby, editors, *ACM CCS 93*, pages 62–73, Fairfax, Virginia, USA, November 3–5, 1993. ACM Press.

[15] Mihir Bellare and Phillip Rogaway. Entity authentication and key distribution. In Douglas R. Stinson, editor, *CRYPTO'93*, volume 773 of *LNCS*, pages 232–249, Santa Barbara, CA, USA, August 22–26, 1994. Springer, Heidelberg, Germany.

[16] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Heidelberg, Germany.

[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, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 619–650, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[18] Mihir Bellare and Björn Tackmann. The multi-user security of authenticated encryption: AES-GCM in TLS 1.3. In Matthew Robshaw and Jonathan Katz, editors, *CRYPTO 2016, Part I*, volume 9814 of *LNCS*, pages 247–276, Santa Barbara, CA, USA, August 14–18, 2016. Springer, Heidelberg, Germany.

[19] Mihir Bellare and Bennet Yee. Forward-security in private-key cryptography. In Marc Joye, editor, *Topics in Cryptology — CT-RSA 2003*, pages 1–18, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg.

[20] Mihir Bellare and Bennet S. Yee. Forward-security in private-key cryptography. In Marc Joye, editor, *CT-RSA 2003*, volume 2612 of *LNCS*, pages 1–18, San Francisco, CA, USA, April 13–17, 2003. Springer, Heidelberg, Germany.

[21] Alexandra Boldyreva, Jean Paul Degabriele, Kenneth G. Paterson, and Martijn Stam. Security of symmetric encryption in the presence of ciphertext fragmentation. In David Pointcheval and Thomas Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 682–699, Cambridge, UK, April 15–19, 2012. Springer, Heidelberg, Germany.

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

[23] 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*, pages 51–62, Chicago, Illinois, USA, October 17–21, 2011. ACM Press.

[24] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145, Las Vegas, NV, USA, October 14–17, 2001. IEEE Computer Society Press.

[25] Ran Canetti, Shai Halevi, and Jonathan Katz. A forward-secure public-key encryption scheme. *Journal of Cryptology*, 20(3):265–294, July 2007.

[26] Ran Canetti and Hugo Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In Birgit Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 453–474, Innsbruck, Austria, May 6–10, 2001. Springer, Heidelberg, Germany.

[27] Ran Canetti and Hugo Krawczyk. Universally composable notions of key exchange and secure channels. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 337–351, Amsterdam, The Netherlands, April 28 – May 2, 2002. Springer, Heidelberg, Germany.

[28] K. Cohn-Gordon, C. Cremers, and L. Garratt. On post-compromise security. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 164–178, June 2016.

[29] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. A formal security analysis of the Signal messaging protocol. In *Proc. IEEE European Symposium on Security and Privacy (EuroS&P) 2017*. IEEE, April 2017. To appear.

[30] Yvo Desmedt and Yair Frankel. Threshold cryptosystems. In Gilles Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 307–315, Santa Barbara, CA, USA, August 20–24, 1990. Springer, Heidelberg, Germany.

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

[32] Yevgeniy Dodis, Jonathan Katz, Shouhuai Xu, and Moti Yung. Key-insulated public key cryptosystems. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 65–82, Amsterdam, The Netherlands, April 28 – May 2, 2002. Springer, Heidelberg, Germany.

[33] Yevgeniy Dodis, Jonathan Katz, Shouhuai Xu, and Moti Yung. Strong key-insulated signature schemes. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 130–144, Miami, FL, USA, January 6–8, 2003. Springer, Heidelberg, Germany.

[34] Yevgeniy Dodis, Weiliang Luo, Shouhuai Xu, and Moti Yung. Key-insulated symmetric key cryptography and mitigating attacks against cryptographic cloud software. In Heung Youl Youm and Yoojae Won, editors, *ASIACCS 12*, pages 57–58, Seoul, Korea, May 2–4, 2012. ACM Press.

[35] F. Betül Durak and Serge Vaudenay. Generic round-function-recovery attacks for Feistel networks over small domains. Cryptology ePrint Archive, Report 2018/108, 2018. https://eprint.iacr.org/2018/108.

[36] Trevor Perrin (editor) and Moxie Marlinspike. The double ratchet algorithm. https://whispersystems.org/docs/specifications/doubleratchet/, November 20, 2016. Accessed: 2017-06-03.

[37] Facebook. Messenger secret conversations technical whitepaper. Technical whitepaper, https://fbnewsroomus.files.wordpress.com/2016/07/secret_conversations_whitepaper-1.pdf, July 8, 2016. Accessed: 2016-09-15.

[38] 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*, pages 1193–1204, Scottsdale, AZ, USA, November 3–7, 2014. ACM Press.

[39] Marc Fischlin, Felix Günther, Giorgia Azzurra Marson, and Kenneth G. Paterson. Data is a stream: Security of stream-based channels. In Rosario Gennaro and Matthew J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 545–564, Santa Barbara, CA, USA, August 16–20, 2015. Springer, Heidelberg, Germany.

[40] Craig Gentry and Alice Silverberg. Hierarchical ID-based cryptography. In Yuliang Zheng, editor, *ASIACRYPT 2002*, volume 2501 of *LNCS*, pages 548–566, Queenstown, New Zealand, December 1–5, 2002. Springer, Heidelberg, Germany.

[41] Matthew D. Green and Ian Miers. Forward secure asynchronous messaging from puncturable encryption. In *2015 IEEE Symposium on Security and Privacy*, pages 305–320, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press.

[42] Christoph G. Günther. An identity-based key-exchange protocol. In Jean-Jacques Quisquater and Joos Vandewalle, editors, *EUROCRYPT'89*, volume 434 of *LNCS*, pages 29–37, Houthalen, Belgium, April 10–13, 1990. Springer, Heidelberg, Germany.

[43] Felix Günther and Sogol Mazaheri. A formal treatment of multi-key channels. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 587–618, Santa Barbara, CA, USA, August 20–24, 2017. Springer, Heidelberg, Germany.

[44] Joseph Jaeger and Igors Stepanovs. Optimal channel security against fine-grained state compromise: The safety of messaging. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 33–62, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[45] Daniel Jost, Ueli Maurer, and Marta Mularczyk. Efficient ratcheting: Almost-optimal guarantees for secure messaging. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 159–188, Darmstadt, Germany, May 19–23, 2019. Springer, Heidelberg, Germany.

[46] Hugo Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In Joe Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 310–331, Santa Barbara, CA, USA, August 19–23, 2001. Springer, Heidelberg, Germany.

[47] Adam Langley. Pond. GitHub repository, README.md, https://github.com/agl/pond/commit/7bb06244b9aa121d367a6d556867992d1481f0c8, 2012. Accessed: 2017-06-03.

[48] Moxie Marlinspike. Advanced cryptographic ratcheting. https://whispersystems.org/blog/advanced-ratcheting/, November 26, 2013. Accessed: 2016-09-15.

[49] Giorgia Azzurra Marson and Bertram Poettering. Security notions for bidirectional channels. *IACR Trans. Symm. Cryptol.*, 2017(1):405–426, 2017.

[50] Maurice Mignotte. How to share a secret? In Thomas Beth, editor, *EUROCRYPT'82*, volume 149 of *LNCS*, pages 371–375, Burg Feuerstein, Germany, March 29 – April 2, 1983. Springer, Heidelberg, Germany.

[51] Chanathip Namprempre. Secure channels based on authenticated encryption schemes: A simple characterization. In Yuliang Zheng, editor, *ASIACRYPT 2002*, volume 2501 of *LNCS*, pages 515–532, Queenstown, New Zealand, December 1–5, 2002. Springer, Heidelberg, Germany.

[52] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of

LNCS, pages 111–126, Santa Barbara, CA, USA, August 18–22, 2002. Springer, Heidelberg, Germany.

[53] Tatsuaki Okamoto and David Pointcheval. The gap-problems: A new class of problems for the security of cryptographic schemes. In Kwangjo Kim, editor, *PKC 2001*, volume 1992 of *LNCS*, pages 104–118, Cheju Island, South Korea, February 13–15, 2001. Springer, Heidelberg, Germany.

[54] Open Whisper Systems. Signal protocol library for java/android. GitHub repository, https://github.com/WhisperSystems/libsignal-protocol-java, 2017. Accessed: 2017-06-03.

[55] Rafail Ostrovsky and Moti Yung. How to withstand mobile virus attacks (extended abstract). In Luigi Logrippo, editor, *10th ACM PODC*, pages 51–59, Montreal, QC, Canada, August 19–21, 1991. ACM.

[56] Bertram Poettering and Paul Rösler. Towards bidirectional ratcheted key exchange. In Hovav Shacham and Alexandra Boldyreva, editors, *CRYPTO 2018, Part I*, volume 10991 of *LNCS*, pages 3–32, Santa Barbara, CA, USA, August 19–23, 2018. Springer, Heidelberg, Germany.

[57] Phillip Rogaway. Authenticated-encryption with associated-data. In Vijayalakshmi Atluri, editor, *ACM CCS 2002*, pages 98–107, Washington, DC, USA, November 18–22, 2002. ACM Press.

[58] Phillip Rogaway. Nonce-based symmetric encryption. In Bimal K. Roy and Willi Meier, editors, *FSE 2004*, volume 3017 of *LNCS*, pages 348–359, New Delhi, India, February 5–7, 2004. Springer, Heidelberg, Germany.

[59] Phillip Rogaway and Thomas Shrimpton. A provable-security treatment of the key-wrap problem. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 373–390, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Heidelberg, Germany.

[60] Adi Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, November 1979.

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

[62] Victor Shoup. A proposal for an iso standard for public key encryption. Cryptology ePrint Archive, Report 2001/112, 2001. https://eprint.iacr.org/2001/112.

[63] Martin Tompa and Heather Woll. How to share a secret with cheaters. *Journal of Cryptology*, 1(2):133–138, June 1988.

[64] Nik Unger, Sergej Dechand, Joseph Bonneau, Sascha Fahl, Henning Perl, Ian Goldberg, and Matthew Smith. SoK: Secure messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press.

[65] Serge Vaudenay. Secure communications over insecure channels based on short authenticated strings. In Victor Shoup, editor, *CRYPTO 2005*, volume 3621 of *LNCS*, pages 309–326, Santa Barbara, CA, USA, August 14–18, 2005. Springer, Heidelberg, Germany.

[66] WhatsApp. Whatsapp encryption overview. Technical white paper, https://www.whatsapp.com/security/WhatsApp-Security-Whitepaper.pdf, April 4, 2016. Accessed: 2016-09-15.

[67] WhatsApp Blog. Connecting one billion users every day. https://blog.whatsapp.com/10000631/Connecting-One-Billion-Users-Every-Day, July 26, 2017. Accessed: 2018-02-12.