Optimally Secure Instant Messaging with Immediate Decryption and Backup Solutions

J.A.M. Pijnenburg

Department of Information Security
Royal Holloway, University of London

This dissertation is submitted for the degree of
Doctor of Philosophy

March 2023

I would like to dedicate this thesis to everyone who was not granted the privilege of schooling.

Declaration

These doctoral studies were conducted under the supervision of Professor Keith Martin.

I hereby declare that except where specific reference is made to the work of others, the contents of this thesis are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This thesis is the result of my own research, in collaboration with others, whilst enrolled in the Department of Information Security as a candidate for the degree of Doctor of Philosophy. This thesis contains fewer than 100,000 words including references and footnotes.

The contents of Chapter 2 and Chapter 3 are joint work with Bertram Poettering and the content of Chapter 4 is joint work with Gareth Davies. While I designed the protocol presented in Chapter 3, the implementation in C has been provided by Bertram Poettering.

J.A.M. Pijnenburg
March 2023

Acknowledgements

I would like to thank my teacher Jo van der Doelen for posing challenging problems beyond the curriculum to put us in our place every time we got complacent, my supervisor Peter Borm for introducing me to what research actually is, my supervisor Steven Galbraith for hosting me and developing my cryptographic skills, and of course my supervisor Tanja Lange, without whom I would have never embarked on this PhD journey.

I am thankful to my supervisor Kenneth Paterson for his guidance, my supervisor Bertram Poettering for agreeing to take me under his wing (provided I would never try to speak German again), my supervisor Keith Martin for seeing me through the end and my advisor Elizabeth Quaglia for always being warm and kind. I would also like to thank Gareth Davies, whom it has been a pleasure to work with during the pandemic when other research interaction dwindled, Martin Albrecht and Benjamin Dowling for agreeing to examine this thesis and providing helpful comments and remarks to improve its clarity and quality, and special thanks to Claire Hudson for always being there to help me navigate the university's administrative system.

My gratitude to the local Egham residents for being so helpful and accepting me into their community when I moved here — you know who you all are. I have had a great time in this leafy town.

Thanks to Marcel Armour for being on this journey together from start to end, Eamonn Postlethwaite for acting the role of wise old mentor, Alpesh Bhudia, Jodie Knapp and Liam Medley for being such a welcoming family and excellent housemates. Thanks to Erin Hales, Lenka Mareková, Simon-Philipp Merz and Joe Rowell for all the fun trips we went on together, Tabitha Ogilvie for all the times you invited me over for dinner to chat, and Balázs Mezei for all the late nights in the office together catching up on the day's work, eating cheese, discussing the latest developments in the world and always checking in on me when times were rough, but I will never forgive you for convincing me that writing up would only take three weeks.

I am grateful to Lisa Wright for providing me unique insight into British culture, supplying me with Tim Tams and coffee, offering excellent advice and generally being the most fabulous friend one could hope for. Finally, I want to thank my family. My

wife for providing me the impetus to finish my thesis quickly after I languished through the pandemic, for her unfaltering support and for being understanding that some weeks of the PhD life can be very intense and stressful. My sister for never turning down an opportunity to visit (me in) London. My brother and his wife for always providing a space for me where I can be myself. My parents for being so supportive from the start, encouraging us to go to university when they did not have the opportunity themselves, the invaluable life lessons and spoiling me when I come home for Christmas.

Abstract

This thesis explores security goals in the instant messaging setting and backup solutions for messages and media, both in the case where the device is still available (but has limited storage capacity) and the case where the device is no longer accessible (e.g. lost or stolen). For each topic we provide game based security definitions, constructions and security proofs.

In Chapter 2 we design a new ratcheting protocol. Ratcheting protocols let parties securely exchange messages in environments in which state exposure attacks are anticipated. While, unavoidably, some promises on confidentiality and authenticity cannot be upheld once the adversary obtains a copy of a party's state, ratcheting protocols aim at confining the impact of state exposures as much as possible. In particular, such protocols provide *forward secrecy* (after state exposure, past messages remain secure) and *post-compromise security* (after state exposure, participants auto-heal and regain security).

We carefully revisit the security notions for ratcheting in the Immediate Decryption (ID) setting, and introduce the progression of *physical time*, which is new for academic treatments of ratcheting protocols. This allows for formally requiring that (undelivered) ciphertexts automatically *expire* after a configurable amount of time.

In Chapter 3 we put forward a symmetric encryption primitive tailored towards a specific application: outsourced storage. The setting assumes a memory-bounded computing device that inflates the amount of volatile or permanent memory available to it by letting other (untrusted) devices hold encryptions of information that they return on request. We develop the cryptographic mechanism that should be used to achieve confidential and authentic data storage in the encrypt-to-self setting, i.e., where encryptor and decryptor coincide and constitute the only entity holding keys. We argue that standard authenticated encryption represents only a suboptimal solution for preserving confidentiality, as much as message authentication codes are suboptimal for preserving authenticity. The crucial observation is that such schemes instantaneously give up on *all* security promises in the moment the key is compromised. In contrast, data protected with our new primitive remains fully integrity protected and unmalleable.

We investigate in Chapter 4 how users of instant messaging services can acquire strong encryption keys to back up their messages and media with strong cryptographic guarantees, without any long-term secret storage. Extending the end-to-end encryption guarantees from just message communication to also incorporate backups has so far required either some trust in an instant messaging or outsourced storage provider, or use of costly third-party encryption tools with unclear security guarantees. Recent works have proposed solutions for password-protected key material, however all require one or more servers to generate and/or store per-user information, inevitably invoking a cost to the users.

We define *distributed key acquisition* (DKA) as the primitive for the task at hand, where a user interacts with one or more servers to acquire a strong cryptographic key, and both user and server store as little as possible. We present a construction framework that we call PERKS—Password-based Establishment of Random Keys for Storage—providing efficient, modular and simple protocols that utilize Oblivious Pseudorandom Functions (OPRFs) in a distributed manner with minimal storage by the user (just the password) and servers (a single global key for all users).

Table of contents

List of figures

List of figures

List of tables

Chapter 1

Introduction

Instant messaging allows two (or more) parties to communicate in near real time by transmitting data via the Internet (or more generally any computer network both parties are connected to). The IRC protocol was the first to achieve widespread adoption, but with the rise of smartphones, first BlackBerry Messenger and later WhatsApp became a dominant player in the market. Most modern instant messaging applications are not limited to text messaging and offer many features such as images, GIFs, reactions, stickers, read receipts, file sharing and voice/video calling. We abstract all of this away and when we write 'sending a message' in this thesis, we refer to the data representing any of the above. Technically, the instant messaging protocol can be peer-to-peer, but popular applications follow a client-server model, where the instant messaging provider accepts messages from the sender and subsequently pushes them to the receiver. This way users do not have to be online simultaneously to be able to exchange messages.

While many applications encrypt data in transit from the sender to the server and from the server to the receiver, end-to-end encryption is less common. With end-to-end encryption we mean the data is encrypted with keys only the endpoints (e.g. each party's mobile phone) have access to. This prevents the instant messaging provider from reading the contents of the messages and thus from reporting users to authoritarian regimes. It should be obvious that this is still extremely relevant today, but examples include the blocking and deletion of WeChat accounts associated with China's LGBTQ movement in July 2021[1] and data experts predict that in the USA call histories, text messages and emails will be used to prosecute anyone seeking reproductive care, anyone assisting them and anyone providing it.[2] There are currently many instant messaging

[1] https://www.bbc.co.uk/news/world-asia-china-57759480

[2] https://www.eff.org/deeplinks/2022/05/what-companies-can-do-now-protect-digital-rights-post-roe-world

applications available that provide end-to-end encryption including but not limited to Signal, Whatsapp and iMessage. Telegram is another popular application promising encrypted messaging, but by default chats are only encrypted to their server, rather than end-to-end. While there is an option to enable end-to-end encryption for individual chats, there is no such possibility for group chats. Moreover, it relies on unstudied assumptions and has been susceptible to attacks [2].

Users usually download an instant messaging application when they acquire a new phone and then use it for a long time, i.e. many months. Because of this nature, and the fact that mobile phones are easily lost, stolen or confiscated, the resilience of messaging protocols against state exposure is considered crucial. In such an attack, the adversary obtains a copy of the user's state. Now the goal of secure instant messaging is to minimize the effect of such an exposure. In the trivial case where the same key is used for every message, such an attack will be devastating: the adversary will be able to decrypt every message ever sent in the past and every message that will be sent in the future. It is intuitive that for optimal security we will want to use a unique encryption key for every message. The challenge arises how to update our user state in a way such that an adversary cannot use it to acquire past encryption keys nor to obtain future encryption keys.

We remark that end-to-end encryption only focuses on protecting the contents of the message and that there still is a vast trove of metadata available. This includes for example the timing, length and frequency of the messages, location data and the message recipient(s). Knowing that a person just messaged someone can already be very revealing, for example contacting a newspaper may raise questions in itself, without the content being known. It also allows analysts to build a social web of who is talking to whom and to identify a person's friends.

1.1 Organization

This thesis explores three interesting problems in the instant messaging setting. We first focus on the core problem of how to securely exchange messages. Ratcheting protocols play a crucial part here to achieve optimally secure instant messaging. Hence, we provide an informal high level introduction to the concept of ratcheting in Sec. 1.4, expanding on the abstract. For a more formal and detailed treatment we refer the reader to Chapter 2. The second problem we turn our attention to is outsourced storage. After all, it makes little sense to focus on optimal security for the exchange of messages if users subsequently upload the plaintext to the cloud to backup their chats. While standard

cryptographic tools can be used to encrypt your data, we investigate the setting where the encryptor and decryptor are the same party. In Chapter 3 we design a surprisingly efficient provably secure construction that remains fully integrity protected even when the key is compromised. Unsurprisingly, a user may want to access their backup if they lose access to their device (and the key stored on it). Anecdotally, this even appears to be the most common usage pattern. However, now we have encrypted our backup, we need a way to recover the (decryption) key. Thus, the focus of Chapter 4 is how to generate (random) keys such that they can be recovered without trusting a third party key escrow server nor requiring the user to save a backup of their key.

Further, in this introduction, we define the notation that will be used throughout this thesis, some of which may be non-standard. Subsequently we define the syntax of several standard cryptographic primitives, which will be employed throughout the remainder of this thesis and will be assumed to be well understood.

1.2 Contents

This thesis contains adaptations of the following four publications.

1. Jeroen Pijnenburg and Bertram Poettering. "Encrypt-to-Self: Securely Outsourcing Storage". In: *ESORICS 2020, Part I*. ed. by Liqun Chen et al. Vol. 12308. LNCS. Springer, Heidelberg, Sept. 2020, pp. 635–654. DOI: 10.1007/978-3-030-58951-6_31.

2. Jeroen Pijnenburg and Bertram Poettering. "Efficiency Improvements for Encrypt-to-Self". In: *CYSARM@CCS*. ACM, Nov. 2020, pp. 13–23. DOI: 10.1145/3411505.3418438.

3. Gareth T. Davies and Jeroen Pijnenburg. "PERKS: Persistent and Distributed Key Acquisition for Secure Storage from Passwords". In: *SAC 2022*. LNCS. Springer, Heidelberg, Aug. 2022, To Appear.

4. Jeroen Pijnenburg and Bertram Poettering. "On Secure Ratcheting with Immediate Decryption". In: *ASIACRYPT 2022, Part III*. ed. by Shweta Agrawal and Dongdai Lin. Vol. 13793. Lecture Notes in Computer Science. Springer, Dec. 2022, pp. 89–118. DOI: 10.1007/978-3-031-22969-5_4.

The following publication was also written during my studies at Royal Holloway, University of London.

5. Jeroen Pijnenburg and Bertram Poettering. "Key Assignment Schemes with Authenticated Encryption, revisited". In: *IACR Trans. Symm. Cryptol.* 2020.2 (2020), pp. 40–67. ISSN: 2519-173X. DOI: 10.13154/tosc.v2020.i2.40-67.

My research was supported by the EPSRC and the UK government as part of the Centre for Doctoral Training in Cyber Security at Royal Holloway, University of London (EP/P009301/1).

1.3 Notation

For the Boolean constants True and False we write `T` and `F`, respectively. We let $\mathbb{N} = \{0,1,\ldots\}$ and $\mathbb{N}^+ = \{1,2,\ldots\}$. For $a,b \in \mathbb{N}$, we define $\Delta(a,b) := |b-a|$. If $a < b$, we use notations $[a\,..\,b] = \{a,\ldots,b\}$, $[b] = [0\,..\,b]$, $[\![a\,..\,b]\!] = [a\,..\,(b-1)]$, and $[\![b]\!] = [0\,..\,(b-1)]$ (where $[\![0]\!]$ is the empty set). We further write $[\![\infty]\!]$ for the set of natural numbers $\mathbb{N} = \{0,\ldots\}$.

We specify scheme algorithms and security games in pseudocode. In such code we write '$var \leftarrow exp$' for evaluating expression exp and assigning the result to variable var. Here, expression exp may comprise the invocation of algorithms.[3] If var is a set variable and exp evaluates to a set, we write $var \xleftarrow{\cup} exp$ shorthand for $var \leftarrow var \cup exp$ and $var \xleftarrow{\cap} exp$ shorthand for $var \leftarrow var \cap exp$. If S is a finite set, $var \leftarrow_{\$} S$ stands for picking an element of S uniformly at random and assigning the result to variable var. Associative arrays implement the 'dictionary' data structure: Once the instruction $A[\cdot] \leftarrow exp$ initialized all items of array A to the default value exp, with $A[idx] \leftarrow exp$ and $var \leftarrow A[idx]$ individual items indexed by expression idx can be updated or extracted. A vector variable can be appended to another with the concatenation operator $\shortparallel$, and we will write $var \xleftarrow{\shortparallel} exp$ shorthand for $var \leftarrow var \shortparallel exp$. Note that even if both variables hold strings, concatenation does not mean string concatenation, i.e.: $"a" \shortparallel "b" \neq "ab"$. For a vector variable var we denote with $\mathrm{size}(var)$ the number of defined elements, i.e. elements that are not $\perp$, which may be less than the length of var.

An alphabet Σ is any finite set of symbols or characters. We denote with Σ^n the set of strings of length n and with $\Sigma^{\leq n}$ the strings of length up to (and including) n. In the practical parts of this thesis we assume that $|\Sigma| = 256$, i.e., that all strings are byte strings. We denote string concatenation with $||$. If var is a string variable and exp evaluates to a string, we write $var \xleftarrow{||} exp$ shorthand for $var \leftarrow var || exp$. Further, if exp evaluates to a string, we write $var || var' \leftarrow_n exp$ to denote splitting exp such that we

[3] Non-deterministic algorithms are always executed with uniform coins.

assign the first n characters from *exp* to *var* and assign the remainder to *var′*. When we do not need the remainder, we write $var \leftarrow_n exp$ shorthand for $var \parallel dummy \leftarrow_n exp$ and discard *dummy*.

For a stateful algorithm alg we write $y \leftarrow \text{alg}\langle st\rangle(x)$ shorthand for $(st, y) \leftarrow \text{alg}(st, x)$ to denote an invocation that updates the state *st*. Note x or y may be empty, that is, the algorithm takes no input or produces no output (besides the state). Additionally, we do not exclude the possibility the state may remain unchanged. We specify the syntax of an algorithm alg that takes as input $x \in \mathcal{X}$ and outputs $y \in \mathcal{Y}$ by writing $\mathcal{X} \to \text{alg} \to \mathcal{Y}$. For a stateful algorithm with $st \in \mathcal{S}$ we will write $\mathcal{X} \to \text{alg}\langle\mathcal{S}\rangle \to \mathcal{Y}$ shorthand for $\mathcal{S} \times \mathcal{X} \to \text{alg} \to \mathcal{S} \times \mathcal{Y}$.

Security games are parameterized by an adversary, and consist of a main game body plus zero or more oracle specifications. The adversary can query all oracles specified by the game, in any order and any number of times. The execution of a game G starts with the main game body and terminates when a 'Stop with *exp*' instruction is reached. The value of expression *exp* is taken as the outcome of the game and will be 0 or 1. We write $\mathbb{P}[\text{G}(\mathcal{A})]$ for the probability that an execution of G with adversary $\mathcal{A}$ results in 1.[4] When we write a 'realistic adversary', we mean any algorithm that runs in probabilistic polynomial time.

We define macros for specific combinations of game-ending instructions: We write 'Win' for 'Stop with 1' and 'Lose' for 'Stop with 0', and further for a condition C we write 'Require C' for 'If $\neg C$: Lose', 'Penalize C' for 'If C: Lose', 'Reward C' for 'If C: Win', and 'Promise C' for 'If $\neg C$: Win'. These macros emphasize the specific semantics of game termination conditions. For instance, a game may terminate with 'Reward *cond*' in cases where the adversary arranged for a situation—indicated by *cond* resolving to True—that should be awarded a win (e.g., the crafting of a forgery in an authenticity game).

Some of our games have Expose and/or Tick oracles. The former models a state exposure of the indicated participant, i.e., a full copy of their program state is given to the adversary. The latter models the progression of physical time: On each invocation, one time unit passed and a new time unit has begun. In our models, invoking this oracle requires indicating the user whose clock shall be progressed by one tick. This allows for expressing not perfectly synchronized clocks and transmission delays: If the clocks of Alice and Bob are initially synchronized, then Alice sends a message, then Bob's clock ticks three times, and then he receives the message, then the network delay was three time units.

[4] The probability is taken over the random coins of G and $\mathcal{A}$.

We finally draw attention to an important detail of our algorithm and game notation: algorithms are allowed to abort. Here, by abort we mean the case where an algorithm does not generate output according to its syntax specification, but outputs some error indicator instead.[5] For example, an AE decryption algorithm that rejects an unauthentic ciphertext or a randomized signature algorithm that does not have sufficiently many random bits to its disposal. In this thesis, for generality we assume that *any* scheme algorithm may abort. However, rather than explicitly encoding this in syntactical constraints which would heavily clutter the notation, we assume that if an algorithm invokes another algorithm as a subroutine, and the latter aborts, then the former also immediately aborts. We assume the same for game oracles: if an invoked scheme algorithm aborts, then the oracle immediately aborts as well. The oracle outputs a special symbol, specifically $\perp$, as a result of the oracle query to inform the adversary that execution has been aborted. We believe that our way to handle errors implicitly rather than explicitly contributes to obtaining definitions with clean and clear semantics.

1.4 Ratcheting

Ratcheting protocols serve as core components in most modern instant messaging apps, with billions of users per day. A ratcheting protocol assumes a shared initial secret and allows two parties, Alice and Bob, to exchange encrypted messages by producing a new, distinct key for each message. The goal of a ratcheting protocol is to minimize the impact of state exposures on confidentiality and authenticity. In essence, each party derives a new key with every message such that past keys cannot be computed from current ones, i.e. the key 'ratchets' forward. Additionally, each party mixes in fresh randomness such that future keys cannot be computed from current keys. The details of each construction depend on the security targeted and many security models have been proposed in the literature [48, 98, 32, 56, 66, 86, 65].

Most instances, including Signal [77], guarantee *immediate decryption* (ID): Receivers recover and deliver the messages wrapped in ciphertexts immediately when they become available, even if ciphertexts arrive out-of-order and preceding ciphertexts are still missing. This ensures the continuation of sessions in unreliable communication networks, ultimately contributing to a satisfactory user experience. While most academic treatments consider ratcheting without ID, recent work by Alwen *et al.* [5] proposes the first ID-aware security model for ratcheting and a provably secure construction. Unfortunately, as we note, in their protocol a receiver state exposure allows for the decryption of all prior *undelivered*

[5] Other works in the field represent 'abort' with 'output $\perp$'.

ciphertexts. As a consequence, from an adversary's point of view, intentionally preventing the delivery of (a fraction of) the ciphertexts of a conversation, and exposing the receiver (days) later, allows for correctly decrypting all suppressed ciphertexts. The same attack works against Signal. The case of undelivered messages is a very realistic threat as governments in Belarus, Burma, China, Egypt, Ethiopia, India, Iran, Iraq, Libya, Russia, Syria and Venezuela have shown their hand to block traffic for specific applications or even orchestrate a complete shutdown[6] and subsequently arrested activists and protestors, confiscating their devices. We remark it is of course far more important that the sender deletes the plaintext messages on their phone in such situations, before worrying about undelivered ciphertexts.

We argue that the level of (forward-)security realized by the protocol of Alwen *et al.*, and mandated by their security model, is considerably lower than both intuitively expected and technically possible. We carefully revisit the security notions for ratcheting in the ID setting and provide a provably secure construction. One component of our model, that is new for academic treatments of ratcheting protocols, is that it reflects the progression of *physical time*. This allows for formally requiring that (undelivered) ciphertexts automatically *expire* after a configurable amount of time.

1.5 Preliminaries

Our constructions will employ several standard cryptographic primitives. Among others, we require a one-time IND-CPA secure symmetric encryption scheme, a strongly unforgeable signature scheme and an IND-CCA secure KEM. We provide the syntax and security definitions of these primitives here.

1.5.1 Symmetric Encryption

Syntax of symmetric encryption. A *symmetric encryption scheme* for a message space $\mathcal{M}$ consists of a key space $\mathcal{K}$, a ciphertext space $\mathcal{C}$, and algorithms enc, dec with APIs

$$\mathcal{K} \times \mathcal{M} \to \text{enc} \to \mathcal{C} \qquad \mathcal{K} \times \mathcal{C} \to \text{dec} \to \mathcal{M} \ .$$

We say that algorithm dec accepts if it terminates normally (outputting a message); otherwise, if it aborts, we say it rejects. For correctness we require that for all $k \in \mathcal{K}$ and $m \in \mathcal{M}$, for all $c \in [\text{enc}(k,m)]$ we have $\text{dec}(k,c) = m$. We assume a one-time

[6] https://www.wired.co.uk/article/internet-shutdowns

Game $\mathrm{CONF}^b(\mathcal{A})$	**Oracle** $\mathrm{Challenge}(m^0, m^1)$
G00 $k \leftarrow_\$ \mathcal{K}$	C00 Require $k \neq \bot$
G01 Invoke $\mathcal{A}$	C01 Require $m^0 \equiv m^1$
G02 Lose	C02 $c \leftarrow \mathrm{enc}(k, m^b)$
Oracle $\mathrm{Decide}(b')$	C03 $k \leftarrow \bot$
D00 Stop with b'	C04 Return c

Fig. 1.1 Games $\mathrm{CONF}^0, \mathrm{CONF}^1$ for one-time IND-CPA secure encryption schemes. Equivalence relation $\equiv$ on $\mathcal{M}$ is defined such that $m^0 \equiv m^1 :\Leftrightarrow |m^0| = |m^1|$.

Game $\mathrm{AUTH}(\mathcal{A})$	**Oracle** $\mathrm{Sign}(m)$	**Oracle** $\mathrm{Vfy}(m, \sigma)$
G00 $\mathrm{SM} \leftarrow \emptyset$	S00 $\sigma \leftarrow \mathrm{sign}(sk, m)$	V00 $\mathrm{vfy}(vk, m, \sigma)$
G01 $(sk, vk) \leftarrow \mathrm{gen}$	S01 $\mathrm{SM} \xleftarrow{\cup} \{(m, \sigma)\}$	V01 If $(m, \sigma) \notin \mathrm{SM}$:
G02 Invoke $\mathcal{A}(vk)$	S02 Return σ	V02 Reward
G03 Lose		

Fig. 1.2 Game AUTH for signature schemes, defining strong unforgeability.

IND-CPA secure encryption scheme. We define the confidentiality notion of one-time indistinguishability under chosen-plaintext attacks via the game in Fig. 1.1. The advantage of adversary $\mathcal{A}$ is defined as $\mathbf{Adv}^{\mathrm{conf}}(\mathcal{A}) := |\mathbb{P}[\mathrm{CONF}^1(\mathcal{A})] - \mathbb{P}[\mathrm{CONF}^0(\mathcal{A})]|$.

1.5.2 Signatures

We briefly recall the syntax of signature schemes and for reference we produce the standard security notion of strong unforgeability (SUF).

Syntax. A *signature* scheme for a message space $\mathcal{M}$ consists of a signing key space $\mathcal{SK}$, a verification key space $\mathcal{VK}$, a signature space Σ, and algorithms $\mathrm{gen}, \mathrm{sign}, \mathrm{vfy}$ with APIs

$$\mathrm{gen} \to \mathcal{SK} \times \mathcal{VK} \qquad \mathcal{SK} \times \mathcal{M} \to \mathrm{sign} \to \Sigma \qquad \mathcal{VK} \times \mathcal{M} \times \Sigma \to \mathrm{vfy} \ .$$

We say that algorithm vfy accepts if it terminates normally; otherwise, if it aborts, we say it rejects. We expect of a correct signature scheme that for all $(sk, vk) \in [\mathrm{gen}]$ and $m \in \mathcal{M}$ and $\sigma \in [\mathrm{sign}(sk, m)]$ we have that $\mathrm{vfy}(vk, m, \sigma)$ accepts.

Strong Unforgeability. We define the authenticity notion of strong unforgeability via the game in Fig. 1.2. The advantage of adversary $\mathcal{A}$ is defined as $\mathbf{Adv}^{\mathrm{auth}}(\mathcal{A}) := \mathbb{P}[\mathrm{AUTH}(\mathcal{A})]$.

1.5.3 Key Encapsulation Mechanisms

We briefly recall the syntax of KEMs and for reference we reproduce the standard security notion of confidentiality against active attacks (IND-CCA),

Syntax. A *key encapsulation mechanism* scheme for a key space $\mathcal{K}$ consists of a secret key space $\mathcal{SK}$, a public key space $\mathcal{PK}$, a ciphertext space $\mathcal{C}$, and algorithms $\mathrm{gen}, \mathrm{enc}, \mathrm{dec}$ with APIs

$$\mathrm{gen} \to \mathcal{SK} \times \mathcal{PK} \qquad \mathcal{PK} \to \mathrm{enc} \to \mathcal{K} \times \mathcal{C} \qquad \mathcal{SK} \times \mathcal{C} \to \mathrm{dec} \to \mathcal{K} \ .$$

We say that algorithm dec accepts if it terminates normally (outputting a key); otherwise, if it aborts, we say it rejects. We expect of a correct KEM that for all $(sk, pk) \in [\mathrm{gen}]$ and $(k, c) \in [\mathrm{enc}(pk)]$ and $k' \in [\mathrm{dec}(sk, c)]$ we have that $k' = k$.

Security of KEM. We define the confidentiality notion of indistinguishability under chosen-ciphertext attacks via the game in Fig. 1.3. The advantage of adversary $\mathcal{A}$ is defined as $\mathbf{Adv}^{\mathrm{conf}}(\mathcal{A}) := |\mathbb{P}[\mathrm{CONF}^1(\mathcal{A})] - \mathbb{P}[\mathrm{CONF}^0(\mathcal{A})]|$.

Game $\mathrm{CONF}^b(\mathcal{A})$	**Oracle** Challenge	**Oracle** $\mathrm{Dec}(c)$
G00 $\mathrm{SC} \leftarrow \emptyset$	C00 $(k^0, c) \leftarrow \mathrm{enc}(pk)$	D10 $k \leftarrow \mathrm{dec}(sk, c)$
G01 $\mathrm{K}[\cdot] \leftarrow \cdot$	C01 $k^1 \leftarrow_{\$} \mathcal{K}$	D11 If $c \in \mathrm{SC}$:
G02 $(sk, pk) \leftarrow \mathrm{gen}$	C02 Promise $c \notin \mathrm{SC}$	D12 Promise $k = \mathrm{K}[c]$
G03 Invoke $\mathcal{A}(pk)$	C03 $\mathrm{SC} \xleftarrow{\cup} \{c\}$	D13 $k \leftarrow \diamond$
G04 Lose	C04 $\mathrm{K}[c] \leftarrow k$	D14 Return k
Oracle $\mathrm{Decide}(b')$	C05 Return (k^b, c)	
D00 Stop with b'		

Fig. 1.3 Games $\mathrm{CONF}^0, \mathrm{CONF}^1$ for KEMs, defining indistinguishability under active attacks. Note that the game also defines functionality [C02,D12].

1.5.4 Secret Sharing Schemes

We now define a secret sharing scheme SSS, that allows an entity to share some secret value k (belonging to some finite set S_1) among n parties, such that any t of the shares enable reconstruction of k, while any set of $\mathrm{t}-1$ shares reveals nothing about k. The exposition here is adapted from Boneh and Shoup [27].

A secret sharing scheme $\mathrm{SSS} = (\mathrm{SecShare}, \mathrm{SecCombine})$ over a finite set S_1 consists of two algorithms. Sharing algorithm $\mathrm{SecShare}(k, \mathrm{t}, \mathrm{n})$ is probabilistic, taking as input $k \in \mathrm{S}_1$ for $0 \leq \mathrm{t} \leq \mathrm{n}$ and returning shares $\vec{\alpha} = \{\alpha_1, \ldots, \alpha_\mathrm{n}\} \in \mathrm{S}_2^\mathrm{n}$. Reconstruction algorithm

SecCombine($\vec{\alpha}'$) is deterministic, taking as input $\vec{\alpha}' = \{\alpha'_1, \ldots, \alpha'_{\mathrm{t}}\} \in \mathrm{S}_2^{\mathrm{t}}$ and returning the reconstructed secret k. Correctness demands that for every secret $k \in \mathrm{S}_1$, every set of n shares $\vec{\alpha}$ output by SecShare(k, t, n), and every subset $\{\alpha'_1, \ldots, \alpha'_{\mathrm{t}}\} = \vec{\alpha}' \subseteq \vec{\alpha}$ of size t, then SecCombine($\vec{\alpha}'$) $= k$.

Definition 1.5.1 (SSS Security)**.** *A secret sharing scheme* (SecShare, SecCombine) *over* S_1 *is secure if for every* $k, k' \in \mathrm{S}_1$*, and every subset* $\vec{\alpha}' \in \mathrm{S}_2^{\mathrm{t}-1}$*, the distribution* SecShare$(k, \mathrm{t}, \mathrm{n})[\vec{\alpha}']$ *is identical to the distribution* SecShare$(k', \mathrm{t}, \mathrm{n})[\vec{\alpha}']$.

The most well known secret sharing scheme is due to Shamir [92] using polynomial interpolation and is suitable for our purposes. The scheme makes use of the fact that a polynomial of degree $t-1$ over a field $\mathbb{F}$ is completely determined by t of its points. A Shamir secret sharing scheme is a secret sharing scheme over $\mathrm{S}_1 = \mathbb{F}_q$ with prime power $q > \mathrm{n}$ and shares are elements of $\mathrm{S}_2 = \mathbb{F}_q^2$. SecShare($k$, t, n) is defined as follows:

1. Choose $a_1, \ldots, a_{t-1} \leftarrow_{\$} \mathbb{F}_q$ and define the polynomial

$$f(x) \coloneqq a_{t-1}x^{t-1} + a_{t-2}x^{t-2} + \ldots + a_1 x + k \in \mathbb{F}_q[x].$$

2. Choose n non-zero points $x_1, \ldots, x_n \in \mathbb{F}_q$ and define $\alpha_i = (x_i, f(x_i)) \in \mathbb{F}_q^2$.

3. Output $\vec{\alpha} = \{\alpha_1, \ldots, \alpha_n\}$.

SecCombine(α) interpolates the polynomial f from the input of t points that completely determine it and outputs $k = f(0)$. A simple method to interpolate the polynomial is called Lagrange interpolation. We refer to [27] for a detailed specification of Lagrange interpolation and a security proof of Shamir's secret sharing scheme. In this thesis we choose S_1 such that it matches our key space $\mathcal{K}$, for example $\mathrm{S}_1 = \mathbb{F}_{2^{256}}$ if we have a 256-bit key space.

Chapter 2

Optimally Secure Ratcheting with Immediate Decryption

We consider a communication model between two parties, Alice and Bob, as it occurs in real-world instant messaging (e.g., in smartphone-based apps like Signal). A key principle in this context is that the parties are only very loosely synchronized. For instance, a "ping-pong" alteration of the sender role is not assumed but parties can send concurrently, i.e., whenever they want to. Further, specifically in phone-based instant messaging, a generally unpredictable network delay has to be tolerated: While some messages are received split seconds after they are sent, it may happen that other messages are delivered only with a considerable delay.[1] We refer to this type of communication (with no enforced structure and arbitrary network delays) as *asynchronous*. We say that asynchronous communication has *in-order* delivery if messages always arrive at the receiver in the order they were sent (what Alice sends first is received by Bob before what she sends later); otherwise, if in-order delivery cannot be guaranteed by the network, we say that the communication has *out-of-order* delivery.

2.1 Introduction

The central cryptographic goals in instant messaging are that the confidentiality and integrity of messages are maintained. As communication sessions are routinely long-lived (e.g., go on for months), and as mobile phones are so easily lost, stolen, confiscated, etc., the resilience of solutions against *state exposure attacks* has been accepted as pivotal. In such an attack, the adversary obtains a full copy of the attacked user's program

[1] E.g., delays of hours can occur if a phone is switched off over night or during a long-distance flight.

state.[2] We say that a protocol provides *forward secrecy* if after a state exposure the already exchanged messages remain secure (in particular confidential), and we say that it provides *post-compromise security* if after a state exposure the attacked participant heals automatically and regains full security if the adversary remains passive.

Past research efforts succeeded with proposing various security models and constructions for the (in-order) asynchronous communication setting with state exposures [48, 98, 32, 56, 66, 86, 65]. The rule of thumb "the stronger the model the more costly the solution" applies also to the ratcheting domain, and the indicated works can be seen as positioned at different points in the security-vs-cost space. For instance, the security models of [56, 86] are the strongest (for excluding no attacks beyond the trivial ones) but seem to necessitate HIBE-like building blocks [12], while [32, 48, 66] work with a relaxed healing requirement (either parties do not recover completely or recovery is delayed) that can be satisfied with DH-inspired constructions.

While the works discussed above exclusively consider communication with in-order delivery, popular instant-messaging solutions like Signal are specifically designed to tolerate out-of-order delivery [77, Sec. 2.6] in order to best deal with the needs of users who want to effectively communicate despite temporary network outages, radio dead spots, etc. Given this means that the protocols cannot rely on ciphertexts arriving in the order they were sent, let alone that they arrive at all, the *immediate decryption* (**ID**) property of such protocols demands that independently of the order in which ciphertexts are received, and independently of the ciphertexts that might still be missing, any ciphertext shall be decryptable for immediate display in the moment it arrives.[3] The ID property received academic attention only recently, in an article by Alwen, Coretti, and Dodis (**ACD**) [5]. As the authors point out, while virtually all practical secure messaging solutions do support ID, most rigorous treatments do not. The work of ACD aims at closing this gap, and we revisit and refine their results.

We provide more details about the work of ACD and explain how we improve upon it. Their main focus is on the Double Ratchet (**DR**) primitive which is one of the core components of the Signal protocol [77, 40]. DR was specifically developed to allow for simultaneously achieving forward and post-compromise security in ID-supporting instant messaging. ACD contribute a formal security model for this primitive and detail how instant messaging can be constructed from it. This approach, however, does *not* formally

[2]Program states could leak because of malware executed on the user's phone, by analysing backup images of a phone's memory that are stored insufficiently encrypted in the cloud, by analysing memory residues on swap drives, etc. Less technical conditions include that users are legally or illegally coerced to reveal their states.

[3]In the user interface, placeholders could indicate messages that are still missing.

guarantee that their solution is also secure in an intuitive sense: A user would expect their sent messages to be secure. We identified an attack that should not be successful against a secure ID-supporting instant messaging protocol, yet if applied against the ACD protocol (or Signal) it leads to the full decryption of arbitrarily selected ciphertexts.

Our attack is fairly simple and succeeds as follows:[4] Assume Alice encrypts, possibly spread over a timespan of months, a sequence of messages $m_1, \ldots, m_L$ and sends the resulting ciphertexts $c_1, \ldots, c_L$ to Bob. An adversary who is interested in learning the target message m_1, arranges that all ciphertexts with exception of c_1 arrive at their destination. By the ID property, Bob decrypts the ciphertexts $c_2, \ldots, c_L$ delivered to him and recovers the messages $m_2, \ldots, m_L$. Further, expecting that the missing c_1 is eventually delivered, he remains in the position to eventually decrypt c_1. But if Bob can decrypt c_1, the adversary, after obtaining Bob's key material via a state exposure, can decrypt c_1 as well, revealing the target message m_1. Note that the attack is not restricted to targeting specifically the first ciphertext; it would similarly work against any other ciphertext, or against a selection of ciphertexts, and the adversary would in all cases fully recover the target messages from just one state exposure. That is, for an adversary who wants to learn specific messages of a conversation secured with Signal or the protocol of ACD, it suffices to suppress the delivery of the corresponding ciphertexts and arrange for a state exposure at some later time.

2.1.1 Contributions

Main Conceptual Contribution. Our attack seems to indicate that the immediate decryption (ID) and forward secrecy (FS) goals, by their very nature, are mutually exclusive, meaning that one can have the one or the other, but not both. Our interpretation is less black and white and involves refining both the ID and the FS notions. While we accept that the out-of-order delivery and ID features are necessary to adequately deal with unreliable networks, we argue that it makes sense to also put a cap on the acceptable amount of transmission delay. For concreteness, let threshold δ specify a maximum delay that messages travelling on the network may experience (including when transmissions are less reliable). Then ciphertexts that are sent at a time t_1 and arrive at a time t_2 should be deemed useful and decryptable only if $\Delta(t_1, t_2) \leq \delta$, while they should be considered expired and thus disposable if $\Delta(t_1, t_2) > \delta$. Once a threshold δ on the delay is fixed, the ID notion can be weakened to require the correct decryption of ciphertexts only if the latter are at most δ old, and the FS notion can be weakened

[4] See Sec. 2.3.4 for a formal treatment.

to protect past messages under state exposure only if they are older than δ (or already have been decrypted). As we show, once the two notions have been weakened in this sense, they fit together without contradicting each other. That is, we promote the idea of integrating a notion of progressing physical time into the ID and FS definitions so that their seemingly inherent rivalry is resolved and one can have a trade-off between both properties, parameterized by δ.

Our models and constructions see δ as a configurable parameter. The value to pick depends on the needs of the participants. For instance, if Alice and Bob are political activists operating under an oppressive regime, choosing $\delta < 15$ mins might be useful; more relaxed users might want to choose $\delta = 1$ week. Note that for $\delta = \infty$ our definitions 'degrade' to the no-expiration setting.

Main Technical Contributions. We start with a compact description of our three main technical contributions. We expand on the topics subsequently.

In a nutshell, the contributions of this chapter are: (1) We introduce the concept of evolving physical time to formal treatments of secure messaging. This allows us to express requirements on the automatic expiration of ciphertexts after a definable amount of time. (2) We propose new security models for secure messaging with immediate decryption (ID). As our approach is to have the security definitions disregard the unavoidable trivial attacks but nothing else, our models are particularly strong. By incorporating the progressing of physical time into our notions, our FS and ID definitions are not in conflict with each other. (3) We contribute a proof-of-concept protocol that provably satisfies our security notions. Efficiency-wise our protocol might be less convincing than the ACD protocol and Signal, but it is definitely more secure.

(1) Modelling physical time. Among the many possible approaches to formalizing evolving physical time, the likely most simple option is sufficient for our purposes. In our treatments we assume that participants have access to a local clock device that notifies them periodically through events referred to as *ticks* about the elapse of a configurable amount of time.[5] The clocks of all participants are expected to be configured to the same ticking frequency (e.g., one tick every one minute), but otherwise our synchronization demands are very moderate: The only aspect relevant for us is that when Alice sends a ciphertext at a time t_1 (according to her clock) and Bob receives the ciphertext at a time t_2 (according to his clock), then we expect that the difference $\Delta(t_1, t_2)$ be meaningful to declare ciphertexts fresh or expired. More precisely, we deem ciphertexts with $\Delta(t_1, t_2) \leq \delta$, for a configurable threshold δ, fresh and thus acceptable, while we

[5] Modern computing environments provide such a service right away. For instance, in Linux, via the `setitimer` system call or the `alarm` standard library function.

consider all other ciphertexts expired and thus discardable. Note here that threshold δ specifies both a maximum on the tolerated network delay and on a possibly emerging clock drift between the sender's and the receiver's clock. The right choice of threshold δ is an implementation detail which controls the robustness-security trade-off.[6] See above for a discussion on how to choose δ. We acknowledge that the assumption of loosely synchronized clocks in an adversarial setting is contentious. However, desynchronized clocks would only harm availability: Alice and Bob cannot communicate any further. This is similar to the adversary cutting the wire between them and we remark that no protocol can realistically defend against this as the adversary is assumed to control the network.

(2) Security models. We develop security models for secure messaging with out-of-order delivery and immediate decryption (ID). We claim two improvements over the only prior model in this setting (ACD, [5]): (a) We incorporate physical time into all correctness and security notions. For instance, when formulating the correctness requirements, in contrast to ACD we do not require the correct decryption of expired ciphertexts, and our confidentiality definitions deem state exposure based message recovery attacks successful if the targeted ciphertext is expired. (b) We strengthen the ACD model to the maximum level of attainable security. Recall that ACD was designed for analysing Double Ratchet based constructions which were proven to achieve only limited security already in the in-order delivery setting [56, 86].[7] In contrast to ACD, our models are designed to exclude the unavoidable 'trivial' attacks but nothing else, thus guaranteeing optimal security. (In Sec. 2.3.2 and Sec. 2.3.3 we review examples of such trivial attacks. We also list attacks that are included in our model but excluded by the ACD model.)

(3) Our construction. We propose a proof-of-concept construction that provably satisfies our security definitions. Its cryptographic core is formed by two specialized types of key encapsulation mechanism (KEM): a KeKEM and a KuKEM. In a nutshell, our KeKEM (key-evolving KEM) primitive is a type of KEM where public and secret keys can be linearly updated 'to the next epoch', almost like in forward-secure PKE. In

[6] One might wonder about the resilience of computer clocks against desynchronization attacks where the adversary aims at desynchronizing participants. We note that instant messaging apps are typically run on mobile devices that have access to multiple independent clock sources (e.g., a local clock, NTP, GSM, and GNSS) that can be compared and relied upon when consistent. Only the strongest adversaries can arrange for a common deviation of all these clock sources simultaneously and even in this case our solutions degrade gracefully: The security of our solutions degrade to that of prior proposals only if the clocks of all time sources are stopped completely.

[7] In a nutshell, DR provides optimal security only if used for ping-pong structured communication [56, 86]. In contrast, the constructions of [56, 86] provide security for *any* (in-order) communication pattern, though require stronger primitives than DR.

contrast, our KuKEM (key-updatable KEM) primitive allows for updating keys based on provided auxiliary input strings. In both cases, key updates provide forward secrecy, i.e., 'the updates cannot be undone'. Building further on more standard building blocks like one-time signatures and symmetric authenticated encryption, the keys established by the two KEM types are combined together to finally yield a secure instant messaging protocol. In addition to the cryptographic core, a considerable share of our protocol specification is concerned with data management because the KeKEM and KuKEM primitives require that senders and receivers perform their updates in a strictly synchronized fashion. If ciphertexts arrive out of order, careful bookkeeping is required to let the receiver update in the right order and at the right time.

It is instructive to observe how and where we employ the two types of KEM: Key updates of the KeKEM are closely linked to evolving physical time, while key updates of the KuKEM are related to processing outgoing and incoming ciphertexts. We note that similar KEM variants have been proposed and used in prior work on instant messaging [56, 86, 12], so in this thesis we claim novelty for neither the concepts nor the constructions.

When compared to the constructions of ACD and Signal, our construction is admittedly less efficient, primarily because (a) we employ the KuKEM and KeKEM primitives that seem to require a considerable computational overhead, and (b) the ciphertexts of our protocol are larger. Concerning (a), we note that prior work like [56, 86] that achieves optimal security for the much easier in-order instant messaging case uses the same primitives, and that recent results [12] indicate that their use is actually unavoidable. We conclude from this that the computational overhead that the primitives bring with them seems to represent the due price to pay for the extra security. A similar statement can be made concerning (b): If an instant messaging conversation is such that the sender role strictly alternates between Alice and Bob, then the ciphertext overhead of our protocol, when compared to Signal, is just a couple of bytes per message. If the sender role does not strictly alternate, the ciphertext size grows linearly (with a not too large factor) in the number of messages that the sender still has to confirm to have arrived. Recalling that the non-alternating case is precisely the one where Signal fails to provide optimal security, the ciphertext overhead seems to be fair given the extra security that is achieved.

2.1.2 Related Work

We start with providing a more detailed comparison of our results with those of the prior work mentioned above. We first remark that our results generalize the findings of [56, 86]: If in our models the physical time is 'frozen', messages are always delivered, and

messages are delivered in-order, they express exactly the same security guarantees as [56, 86]. It is clear that as soon as time starts ticking our model is stronger: We allow state exposures once ciphertexts 'expire', while this concept does not exist in [56, 86]. For out-of-order delivery the picture immediately becomes more complicated. Note that when messages are delivered in-order, optimal security demands that user states immediately 'cryptographically diverge'[8] when receiving an unauthentic ciphertext, but for out-of-order delivery the situation becomes more nuanced. Consider the scenario where Alice sends a message and is then state-exposed. Using the obtained state information, the adversary could now trivially and perfectly impersonate Alice towards Bob for the second message. That is, if Bob receives the second ciphertext first, there is no way for him to tell whether it is authentic or not, i.e., to distinguish whether Alice sent or the adversary injected it. If the ciphertext was indeed sent by Alice, correctness would require that Bob remains able to decrypt the first ciphertext. Thus, the latter also has to hold if the ciphertext is unauthentic. Hence, in contrast to the setting with in-order delivery, in the out-of-order setting there are inherent limits to how much the states of Alice and Bob can 'cryptographically diverge' once unauthentic ciphertexts are processed.

Multiple weaker security definitions for secure messaging have been proposed [5, 32, 48, 66]. We provide a brief overview about what makes their security notions suboptimal. In [32, 48] the adversary is forbidden to impersonate a user when a secure key is being established. Hence, in this case the authors do not require recovery from a state exposure (which enables an impersonation attack). In [5, 66] the construction can take longer than strictly necessary to recover from state exposures. This is encoded in the security games by artificially labelling certain win conditions as trivial. Moreover, in both works the user states are not required to immediately 'cryptographically diverge' for future ciphertexts when accepting an unauthentic ciphertext. We note that an important difference between our KuKEM and Healable key-updating Public key encryption (HkuPke) introduced in [66] is that HkuPke key updates are based on secret update information, while our KuKEM is updated with adversarially controlled associated data.

The security definitions of [5, 56, 66] assume a slightly different understanding of what it means to expose a participant. Our understanding is that exposures reveal the current protocol state of a participant to the adversary, while their approach is rather that exposures reveal the randomness used for the next sending operation. The two views seem ultimately incomparable, and likely one can find arguments for both sides. One argument that supports our approach is that modern computing environments have

[8] Meaning: the protocol states, and the key material they encode, become desynchronized beyond recoverability.

RNGs that *constantly* refresh their state based on unpredictable events (e.g., the `RDRAND` instruction of Intel CPUs or the `urandom` device in Linux) so that if one of the situations listed in Footnote 2 leads to a state exposure then it still can be assumed that the randomness used for the next sending operation is indeed safe. A third view considers state exposures to leak a party's state except for signing keys [4], which seems unrealistic.

Our work is not the first to consider a notion of physical time in a cryptographic treatment. See [90] for modelling approaches using linear counters, or [74, 76] for encrypting data 'to the future', or [46] for an article that recognizes that many security features depend on a device knowing the current time and thus proposes an authenticated version of the Network Time Protocol (NTP).

Recent work in the group messaging setting [6] similarly designs their protocol in a modular way and captures security in game based definitions. A main component, *continuous group key agreement* (CGKA) was first defined in [7] and the analysis of [8] shows, even in the passive case, no known CGKA protocol achieves optimal security without using HIBE.

2.1.3 Organization

This article considers the security and constructions of what we refer to as key-updatable bidirectional out-of-order messaging protocols, abbreviated KuBOOM. In Sec. 2.2 we define the security model. In Sec. 2.4 we introduce non-interactive components that we employ in our construction. This includes the mentioned KuKEM and KeKEM primitives. In Sec. 2.5 we finally give our construction.

2.2 Security of KuBOOM Protocols

We formalize KuBOOM protocols. The API consists of five algorithms, where algorithms init, send, and recv (for state initialization, message sending, and message receiving, respectively) are akin to prior work and implement instance initialization, message sending, and message receiving, respectively.[9] Our notion of secure messaging depends on evolving **physical time**; we correspondingly introduce a new algorithm tick that is assumed to be periodically executed by the participants (e.g., once every 10 seconds) and has no visible function beyond updating their internal state, e.g., by incrementing an internal tick counter. Independently of physical time, a notion of **logical time** is

[9] More precisely, our recv algorithm has a dedicated output for reporting to the invoking user which of the priorly sent own messages have been received by the peer; this output does not exist in prior work.

suggested by the sequence in which messages are considered by a sender. We represent both physical and logical time with integer counters: each invocation of tick increments the 'physical counter', and each invocation of send increments the 'logical counter'. The (unique) logical time counter associated with a send invocation is referred to as its **sending index**. The consideration of physical and/or logical time allows for associating ciphertexts with creation **time stamps**. We require that the timestamp information of when a ciphertext is created is embedded, in some form, in the ciphertext. To recover it, we introduce the decoding function ts which, for any ciphertext, indicates both the physical time (with a granularity defined by the tick invocation frequency) and the logical time (i.e., sending index) of the corresponding send invocation.[10] Recovering these values from ciphertexts allows for formulating concepts connected to ciphertext expiration when we define the security notions of KuBOOM.

We proceed with defining the syntax, the semantics (execution environment and correctness), and the security notions associated with KuBOOM protocols.

Syntax. A KuBOOM scheme for a message space $\mathcal{M}$ and an associated-data space $\mathcal{AD}$ consists of a state space $\mathcal{S}$, a ciphertext space $\mathcal{C}$, algorithms $\mathrm{init}, \mathrm{send}, \mathrm{recv}, \mathrm{tick}$, and a decoding function ts. The **initialisation** algorithm init produces (initial) states $st_{\mathtt{A}} \in \mathcal{S}$ and $st_{\mathtt{B}} \in \mathcal{S}$. The **sending** algorithm send takes a state $st \in \mathcal{S}$, an associated-data string $ad \in \mathcal{AD}$, and a message $m \in \mathcal{M}$, and produces an (updated) state $st' \in \mathcal{S}$ and a ciphertext $c \in \mathcal{C}$. The **receiving** algorithm recv takes a state $st \in \mathcal{S}$, an associated-data string $ad \in \mathcal{AD}$, and a ciphertext $c \in \mathcal{C}$, and produces an (updated) state $st' \in \mathcal{S}$, an acknowledgement set $A \subseteq \mathbb{N}$, and a message $m \in \mathcal{M}$. (To Bob, the understanding of output A is that when c was generated by Alice, then Alice had received Bob's ciphertexts with sending index a for all $a \in A$, and vice versa.) The **time progression** algorithm tick takes a state $st \in \mathcal{S}$ and produces an (updated) state $st' \in \mathcal{S}$. The **timestamp decoder** ts is a function that takes a ciphertext $c \in \mathcal{C}$ and recovers a logical timestamp (sending index) $lt \in \mathbb{N}$ and a physical timestamp $pt \in \mathbb{N}$. Thus, if $\mathcal{P}(\mathbb{N})$ denotes the powerset of $\mathbb{N}$, the KuBOOM API is as follows:

$$\mathrm{init} \to \mathcal{S} \times \mathcal{S} \qquad \mathcal{S} \to \mathrm{tick} \to \mathcal{S} \qquad \mathcal{C} \to \mathrm{ts} \to \mathbb{N} \times \mathbb{N}$$

$$\mathcal{AD} \times \mathcal{M} \to \mathrm{send}\langle \mathcal{S} \rangle \to \mathcal{C} \qquad \mathcal{AD} \times \mathcal{C} \to \mathrm{recv}\langle \mathcal{S} \rangle \to \mathcal{P}(\mathbb{N}) \times \mathcal{M}$$

[10] To simplify notation, we model ts as a public function. We believe this is a reasonable choice as adversaries know the recoverable time information anyway. It is straight-forward to adapt our notions to instead support a private ts function (that would take the protocol state as an additional input).

Semantics. We give game based definitions of correctness and security. Recall that the form of secure messaging that we consider supports the out-of-order processing of ciphertexts. This property, of course, has to be reflected in all games, rendering them more complex than those of prior works that deal with easier settings. To manage this complexity, we carefully developed our games such that they share, among each other, as many code lines and game variables as possible. In particular, the games can be seen as derived by individualizing a common **basic game** body in order to express specific aspects of functionality or security. This individualization is done by inserting an appropriate small set of additional code lines.[11] (For instance, the game defining authenticity adds lines of code that identify and flag forgery events.) In the following we explain first the BASIC game and then its refinements FUNC, AUTH, and CONF.[12]

Game BASIC. We first take a quick glance over the BASIC game of Fig. 2.1, deferring the discussion of details to the upcoming paragraphs. The game body [G00–G18] initializes some variables [G00–G13], invokes the init algorithm to initialize states for two users `A` and `B` [G17], and invokes the adversary [G18]. The adversary has access to four oracles, each of which takes an input $u \in \{\mathtt{A}, \mathtt{B}\}$ to specify the targeted user. The Tick oracle gives access to the tick algorithm [T00], the Send oracle gives access to the send algorithm [S00,S07], and the Recv oracle, besides internally recovering the logical and physical sending timestamps of an incoming ciphertext [R00], gives access to the recv algorithm [R01,R29]. Finally, the Expose oracle reveals the current protocol state of a user to the adversary [E06]. The game variables and remaining code lines are related to monitoring the actions of the adversary, allowing for identifying specific game states and tracking the transitions between them. In particular we identified the user-specific states *in-sync* and *authoritative*, the ciphertext properties *sync-preserving*, *sync-damaging*, *certifying*, and *vouching*, and the transitions *losing sync*, *poisoning*, and *healing*, as relevant in the KuBOOM setting. We explain these concepts one by one.

We say that protocol actors are synchronized if their views on the communication is consistent. A little more precisely, a participant Alice is **in-sync** with her peer Bob if all ciphertexts that Alice received are identical with ciphertexts that Bob priorly sent. The complete definition, formalized as part of the BASIC game as discussed below, further requires that the employed associated-data inputs are matching, and that the processing of ciphertexts of an out-of-sync peer also renders the receiver out-of-sync. If Alice is

[11] Removing or modifying existing lines will not be necessary. That said, restricting the options to only add new lines might lead to also introducing a small number of redundancies that could allow for simplifications.

[12] The BASIC game itself is not used to model any kind of functionality or security. It merely describes the execution environment.

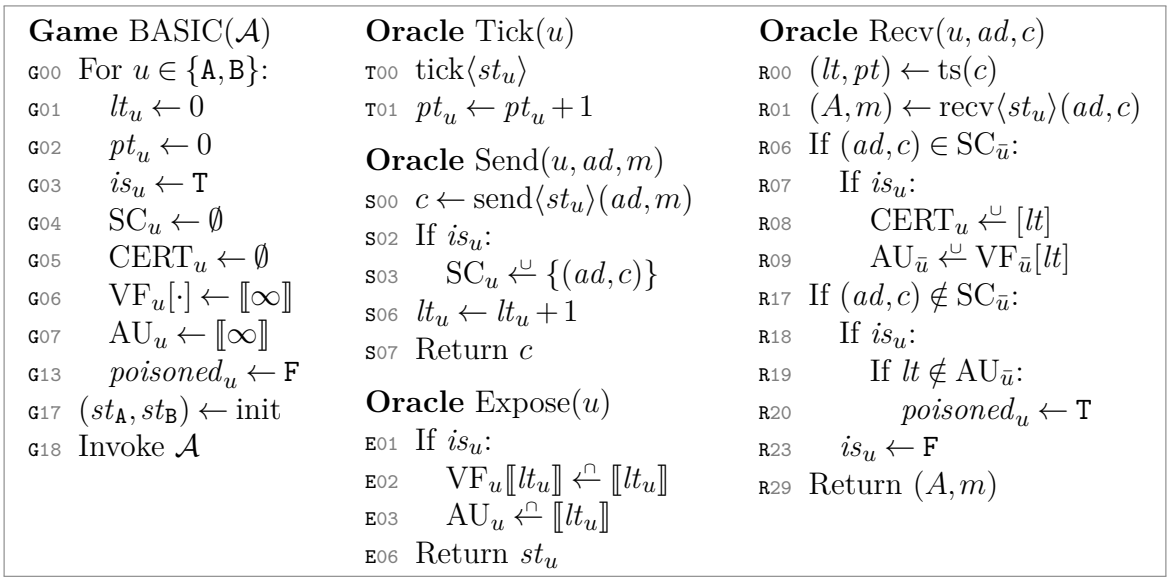

Fig. 2.1 Game BASIC. We refer the reader to Footnote 16 for the interpretation of $\mathrm{VF}_u[\![lt_u]\!]$ in line [E02]. We write $\bar{u}$ for the element such that $\{u, \bar{u}\} = \{\mathtt{A}, \mathtt{B}\}$.

in-sync with Bob, we refer to ciphertexts that Alice can receive without **losing sync** as **sync-preserving**; the ciphertexts that would render her out-of-sync are referred to as **sync-damaging**.[13] See Fig. 2.4 in Sec. 2.3.1 for an example.

As we consider communication algorithms that are stateful, any ciphertext created by a participant may depend on, and may implicitly reflect, the full prior communication history of that participant. That is, if from a sequence of sent ciphertexts only a subset of ciphertexts arrive, then from what *did* arrive the receiver should be able to extract information linked to what was sent before but is still missing. In particular, any ciphertext that is received in-sync should allow for identifying which earlier-sent though later-delivered ciphertexts are authentic. We correspondingly say that in-sync received ciphertexts **certify** the ciphertexts sent *earlier* by the same sender. See Fig. 2.5 in Sec. 2.3.1 for an example.

Ciphertexts can also make promises about the future: Every received ciphertext may carry (cryptographic) information that is used to authenticate later ciphertexts (of the same sender, up to their next exposure). Here we say that ciphertexts (cryptographically) **vouch** for the ones sent *later* by the same participant. See Fig. 2.6 in Sec. 2.3.1 for an example.

[13] The in-sync notion first surfaced in [18] in the context of unidirectional channels. It was extended in [78] to handle bidirectional communication and associated-data strings. Our definitions are based on [78], but adapted to tolerate the out-of-order delivery of ciphertexts.

We finally discuss attack classes that are enabled by exposing the states of users: Once a participant's state becomes known by exposure, it is trivial to impersonate the user, simply by invoking the scheme algorithms with the captured state. We refer to states of a participant as **authoritative** if their actions can *not* be trivially emulated by the adversary in this way. If an impersonation happens right after an exposure, as the adversary can perfectly and permanently emulate all actions of the impersonated party, in addition to all authenticity and confidentiality guarantees being lost, there is also no option to recover into a safe state. We refer to the transition into such a setting, more precisely to the action of exploiting the state exposure of one participant by delivering an impersonating ciphertext to the other participant, as **poisoning** the latter. See Fig. 2.7 in Sec. 2.3.1 for an example. A second option of the adversary after exposing a state is to remain passive (in particular, not to poison the partner). In this case the **healing** property of ratcheting-based secure messaging protocols shall automatically fully restore safe operations. See Fig. 2.8 in Sec. 2.3.1 for an example.

Coming back to the BASIC game of Fig. 2.1, we describe how the above concepts are reflected in the game variables and code lines. We start with the game body [G00–G18]. If $u \in \{\mathtt{A},\mathtt{B}\}$ refers to one of the two participants, integer lt_u ('logical time') reflects the logical time (i.e., next sending index) of u; integer pt_u ('physical time') reflects the physical time of u; Boolean flag is_u ('in-sync') indicates whether u is in-sync with their peer $\bar{u}$; set SC_u ('sent ciphertexts') records the associated-data–ciphertext pairs sent by u; set CERT_u ('certified') indicates which of the peer $\bar{u}$'s sending indices have been certified by receiving an in-sync ciphertext from them; for each sending index lt, set $\mathrm{VF}_u[lt]$ ('vouches for') indicates for which sending indices of u the ciphertext with index lt can vouch for; set AU_u indicates for which sending indices participant u is authoritative; flag $poisoned_u$ indicates whether u was poisoned.

We next explain how these variables are updated throughout the game. The cases of lt_u [G01,S06] and pt_u [G02,T01] are clear. Flag is_u is initialized to `T` [G03], and cleared [R23] in the moment that u receives a ciphertext that the peer $\bar{u}$ either didn't send, or did send but after becoming out-of-sync [R17] (in conjunction with [S02,S03], see next sentence).[14] Set SC_u is initialized empty [G04] and populated [S03] for each sending operation in which u is in-sync [S02].[15] Set CERT_u is initialized empty [G05] and, when a sync-preserving ciphertext is received [R06,R07], populated with all indices prior to, and including, the current one [R08]. All entries of array-of-sets VF_u are initialized to 'all-

[14]The mechanism of considering participants out-of-sync once they process (unmodified) ciphertexts from out-of-sync peers is taken from [78], see Footnote 13.

[15]Note that the sending index of any ciphertext is uniquely recoverable (with function ts), implying that each execution of [S03] adds a new element to the set (collisions cannot occur).

indices' [G06], expressing that, by default, each sending index cryptographically vouches for its entire future (and past). This changes when u's state is exposed, as impersonating u then becomes trivial, and the game reflects this by updating all VF_u entries related to the time preceding the exposure so that the corresponding ciphertexts do not vouch for ciphertexts that are created after the exposure happened [E02].[16] Set AU_u is initialized to 'all-indices' [G07], and indices are removed from it by exposing u's state, and added back to it by letting u heal; more precisely, while exposing u's state removes all indices starting with the current one (marking the entire future as non-authoritative) [E03], receiving a sync-preserving ciphertext from peer $\bar{u}$ [R06,R07] adds the vouched-for entries back [R09] (re-establishing authoritativeness up to the next exposure). Finally, flag $\mathit{poisoned}_u$ is initialized clear [G13], and set [R20] when a sync-damaging ciphertext is received (i.e., one that was not sent by peer $\bar{u}$ [R17] and is the first one making u lose sync [R18]) that was trivially injected after an exposure of peer $\bar{u}$'s state (technically: was crafted for a non-authoritative index [R19]).

This completes the description of the BASIC game. We refine it in the following to obtain three more games, but the basic working mechanisms of the oracles and variables remain the same.

Game FUNC. We specify the expected **functionality** (a.k.a. **correctness**) of a KuBOOM protocol by formulating requirements on how it shall react to receiving valid and invalid ciphertexts. Concretely, in Fig. 2.2 we specify the corresponding FUNC game as an extension of the BASIC game from Fig. 2.1. In the figure, the code lines marked with neither ∘ nor • are taken verbatim from the BASIC game, and the lines marked with ∘ are the ones to be added to obtain the FUNC game. (Ignore the lines marked with • for now.) The FUNC game tests for a total of seven conditions, letting the adversary 'win' if any one of them is not fulfilled. Five of the conditions are checked for all operations (in-sync *and* out-of-sync): The conditions are (1) that the ts decoding function correctly indicates the logical and physical creation time of ciphertexts [S01]; (2) that no sending index is received twice (single delivery of ciphertexts) [G10,R02,R25] (set RI_u records 'received indices'); (3) that expired ciphertexts are not delivered (the reported sender's physical time pt is compared with the receiver's physical time pt_u, tolerating a lag of up to δ time units) [R03]; (4) that physical timestamps increase as logical timestamps do [G11,R04,R26] (set RT_u records 'received timestamps');[17] and (5) that the reported acknowledgement set A never shrinks and never lists never-sent indices [G12,R05,R27]

[16] Line [E02] should be read as 'For all $0 \le i < lt_u$: $\mathrm{VF}_u[i] \leftarrow \mathrm{VF}_u[i] \cap [\![lt_u]\!]$' and expresses that all entries of $\mathrm{VF}_u[\cdot]$ that correspond with prior sending indices are trimmed so that they cover no indices that succeed the current one (including).

[17] A relation $R \subseteq \mathbb{N} \times \mathbb{N}$ is monotone [R04] if for all $(x, y), (x', y') \in R$ we have $x \le x' \Rightarrow y \le y'$.

(set RA_u records 'received acknowledgements'). Two additional conditions are checked for certified ciphertexts (this includes all in-sync ciphertexts, as they certify themselves [R06,R07,R08]): The conditions are (6) that the recv algorithm accurately reports the acknowledgement set A [R13] (recall that set RI_u holds the received indices [G10,R25], allowing to associate this set with each (in-sync) sending operation [G08,S04], so that set $\mathrm{SR}_{\bar{u}}[lt]$ [S04,R13] indicates the indices that participant $\bar{u}$ received from u before $\bar{u}$ used sending index lt in their sending operation); and (7) that encrypted messages are correctly recovered via decryption [G09,S05,R14] (array SM_u records 'sent messages'). We say that a KuBOOM protocol is *functional* if the probability $\mathbb{P}[\mathrm{FUNC}(\mathcal{A})]$ is negligibly small for all realistic adversaries $\mathcal{A}$.

Game AUTH. Our **authenticity** notion focuses on the protection of the integrity of ciphertexts (INT-CTXT). In Fig. 2.2 we specify the corresponding AUTH game as an extension of the BASIC game from Fig. 2.1. In the figure, the code lines marked with neither ∘ nor • are taken verbatim from the BASIC game, and the lines marked with • are the ones to be added to obtain the AUTH game. (This time, ignore the lines marked with ∘.) A KuBOOM scheme provides AUTH security if any adversarial manipulation (or injection) of ciphertexts is detected and rejected. Taking into account that associated-data strings need to be protected in the same vein, as a first approximation the notion could be formalized by adding the instruction 'Reward $is_u \wedge (ad, c) \notin \mathrm{SC}_{\bar{u}}$' to the Recv oracle.[18] Note however that delivering a forged ciphertext to a participant u is trivial if the state of their peer $\bar{u}$ is exposed, and thus a small refinement is due. Recalling that set $\mathrm{AU}_{\bar{u}}$ lists the sending indices for which participant $\bar{u}$ is *authoritative*, i.e., their actions not trivially emulatable, we reward the adversary only if the forgery is made for an index contained in this set [R17,R18,R21]. Recall further that in-sync delivered ciphertexts *certify* prior ciphertexts by the same sender, even if the latter ciphertexts are delivered out-of-sync. In the game we thus reward the adversary also if it forges on a certified index [R17,R22]. We say that a KuBOOM protocol provides *authenticity* if the probability $\mathbb{P}[\mathrm{AUTH}(\mathcal{A})]$ is negligibly small for all realistic adversaries $\mathcal{A}$. We refer the reader to Sec. 2.3.2 for a formalization of the trivial attack excluded by the AUTH game, and an overview of similar but non-trivial attacks that are allowed.

Games $\mathrm{CONF}^0, \mathrm{CONF}^1$. Our **confidentiality** notion is formulated in the style of left-or-right indistinguishability under active attacks (IND-CCA). In Fig. 2.3 we specify corresponding CONF^0 and CONF^1 games. The games are derived from the BASIC game by adding the lines marked with • plus two new oracles: The left-or-right Chal oracle

[18] The instruction should be read as 'Reward the adversary if it makes an in-sync participant accept an associated-data–ciphertext pair for which at least one of associated-data and ciphertext is not authentic'.

Game FUNC($\mathcal{A}$) //with ○
Game AUTH($\mathcal{A}$) //with ●
G00 For $u \in \{\mathtt{A},\mathtt{B}\}$:
G01 $\quad lt_u \leftarrow 0$
G02 $\quad pt_u \leftarrow 0$
G03 $\quad is_u \leftarrow \mathtt{T}$
G04 $\quad \mathrm{SC}_u \leftarrow \emptyset$
G05 $\quad \mathrm{CERT}_u \leftarrow \emptyset$
G06 $\quad \mathrm{VF}_u[\cdot] \leftarrow [\![\infty]\!]$
G07 $\quad \mathrm{AU}_u \leftarrow [\![\infty]\!]$
○ G08 $\quad \mathrm{SR}_u[\cdot] \leftarrow \bot$
○ G09 $\quad \mathrm{SM}_u[\cdot] \leftarrow \bot$
○ G10 $\quad \mathrm{RI}_u \leftarrow \emptyset$
○ G11 $\quad \mathrm{RT}_u \leftarrow \emptyset$
○ G12 $\quad \mathrm{RA}_u \leftarrow \emptyset$
G17 $(st_{\mathtt{A}}, st_{\mathtt{B}}) \leftarrow \mathrm{init}$
G18 Invoke $\mathcal{A}$
G19 Lose

Oracle Tick(u)
T00 $\mathrm{tick}\langle st_u\rangle$
T01 $pt_u \leftarrow pt_u + 1$

Oracle Send(u, ad, m)
S00 $c \leftarrow \mathrm{send}\langle st_u\rangle(ad, m)$
○ S01 Promise $\mathrm{ts}(c) = (lt_u, pt_u)$
S02 If is_u:
S03 $\quad \mathrm{SC}_u \xleftarrow{\cup} \{(ad, c)\}$
○ S04 $\quad \mathrm{SR}_u[lt_u] \leftarrow \mathrm{RI}_u$
○ S05 $\quad \mathrm{SM}_u[lt_u] \leftarrow m$
S06 $lt_u \leftarrow lt_u + 1$
S07 Return c

Oracle Recv(u, ad, c)
R00 $(lt, pt) \leftarrow \mathrm{ts}(c)$
R01 $(A, m) \leftarrow \mathrm{recv}\langle st_u\rangle(ad, c)$
○ R02 Promise $lt \notin \mathrm{RI}_u$
○ R03 Promise $\Delta(pt, pt_u) \leq \delta$
○ R04 Promise $\mathrm{RT}_u \cup \{(lt, pt)\}$ monotone
○ R05 Promise $\mathrm{RA}_u \subseteq A \subseteq [\![lt_u]\!]$
R06 If $(ad, c) \in \mathrm{SC}_{\bar{u}}$:
R07 $\quad$ If is_u:
R08 $\quad\quad \mathrm{CERT}_u \xleftarrow{\cup} [lt]$
R09 $\quad\quad \mathrm{AU}_{\bar{u}} \xleftarrow{\cup} \mathrm{VF}_{\bar{u}}[lt]$
○ R12 $\quad$ If $lt \in \mathrm{CERT}_u$:
○ R13 $\quad\quad$ Promise $A = \mathrm{RA}_u \cup \mathrm{SR}_{\bar{u}}[lt]$
○ R14 $\quad\quad$ Promise $m = \mathrm{SM}_{\bar{u}}[lt]$
R17 If $(ad, c) \notin \mathrm{SC}_{\bar{u}}$:
● R18 $\quad$ If is_u:
● R21 $\quad\quad$ Reward $lt \in \mathrm{AU}_{\bar{u}}$
● R22 $\quad$ Reward $lt \in \mathrm{CERT}_u$
R23 $\quad is_u \leftarrow \mathtt{F}$
○ R25 $\mathrm{RI}_u \xleftarrow{\cup} \{lt\}$
○ R26 $\mathrm{RT}_u \xleftarrow{\cup} \{(lt, pt)\}$
○ R27 $\mathrm{RA}_u \leftarrow A$
R29 Return (A, m)

Oracle Expose(u)
E01 If is_u:
E02 $\quad \mathrm{VF}_u[\![lt_u]\!] \xleftarrow{\cap} [\![lt_u]\!]$
E03 $\quad \mathrm{AU}_u \xleftarrow{\cap} [\![lt_u]\!]$
E06 Return st_u

Fig. 2.2 Games FUNC and AUTH. The FUNC game includes the lines marked with ○ but not the ones marked with ●. The AUTH game includes the lines marked with ● but not the ones marked with ○.

[C00–C08], which behaves similar to the Send oracle but processes one of two possible input messages [C03] depending on bit b that encodes which game CONF^b is played, and the Decide oracle [D00] that lets the adversary control the return value of the game. (A successful adversary manages to correlate this return value with bit b.)

Three new game variables keep track of the actions of the adversary: Variable lx_u ('last exposure', [G14,E04]) indicates the index of the last exposure of user u. Set CH_u ('challenge') represents the set of sending indices for which a challenge query has been

posed for u that peer $\bar{u}$ still should be able to validly decrypt. Indices are added to this set in the Chal oracle [G15,C06], and they are removed from it as a reaction to three events. (1) The corresponding ciphertext becomes invalid because the receiver already processed a ciphertext (the same or a different one) with the same index [R28] (see the corresponding guarantee in the FUNC game [G10,R02,R25]). (2) It becomes invalid because it expired based on physical time: To capture the latter condition we denote with

$$\mathrm{ITC}(u) := \{lt : \exists ad, c, pt \text{ s.t. } (ad, c) \in \mathrm{SC}_{\bar{u}} \wedge \mathrm{ts}(c) = (lt, pt) \wedge \Delta(pt, pt_u) \leq \delta\}$$

('in-time ciphertexts') for participant u the set of sending indices of ciphertexts produced by peer $\bar{u}$ for which the difference between generation time pt and the physical time pt_u of the receiver is less than δ. With the progression of physical time the game removes those indices from set $\mathrm{CH}_{\bar{u}}$ that are not an element of $\mathrm{ITC}(u)$ [T02]. (See the corresponding guarantee in the FUNC game [R03].) (3) Receiving an out-of-sync ciphertext renders u's state incompatible to decrypt *future* challenge queries. Hence all future indices are removed from the challenge set [R24]. Observe this corresponds with [C06]: indices are only added to CH_u for an in-sync peer. Finally, flag xp_u ('exposed') indicates whether the state of u has to be considered known to the adversary after a state exposure. This flag is initially cleared [G16], set when u's state is exposed [E05], and reset if u *heals* by letting peer $\bar{u}$ receive an in-sync ciphertext created after the last exposure [R10,R11].

We next explain how the new variables help identifying four different trivial attack conditions. The first two conditions consider cases where posing a Chal query needs to be prevented because the receiver state is known due to impersonation or exposure: (1) if participant $\bar{u}$'s state was exposed and $\bar{u}$ is impersonated to u, i.e., u is *poisoned*, all future encryptions by u for $\bar{u}$ are trivially decryptable, simply because the adversary can emulate *all* actions of $\bar{u}$ [C01]; (2) encryptions by an in-sync sender u for a state-exposed receiver $\bar{u}$ are trivially decryptable (recall that flag $xp_{\bar{u}}$ traces the latter condition) [C02]. The next condition considers cases where posing an Expose query needs to be prevented because an already made Chal query would become trivial to break: (3) if participant $\bar{u}$ generated (challenge) ciphertext c for u, and the latter should still be able to validly decrypt c, then exposing u makes c trivially decryptable [E00]. The last condition is unrelated to exposures: (4) if participant u in-sync decrypts a ciphertext, by correctness the resulting message is identical to the encrypted message, and thus has to be suppressed by the Recv oracle by overwriting it [R15,R16]. (Note how line R16 corresponds with line R14 of FUNC.) This concludes the description of games CONF^b. We say that a KuBOOM protocol provides *confidentiality* if the difference of probabilities

$|\mathbb{P}[\text{CONF}^1(\mathcal{A})] - \mathbb{P}[\text{CONF}^0(\mathcal{A})]|$ is negligibly small for all realistic adversaries $\mathcal{A}$. We refer the reader to Sec. 2.3.3 for a formalization of the trivial attacks excluded by the CONF game, and similar but non-trivial attacks that are allowed.

Game $\text{CONF}^b(\mathcal{A})$

G00 For $u \in \{\mathtt{A},\mathtt{B}\}$:
G01 $\quad lt_u \leftarrow 0$
G02 $\quad pt_u \leftarrow 0$
G03 $\quad is_u \leftarrow \mathtt{T}$
G04 $\quad \text{SC}_u \leftarrow \emptyset$
G06 $\quad \text{VF}_u[\cdot] \leftarrow [\![\infty]\!]$
G07 $\quad \text{AU}_u \leftarrow [\![\infty]\!]$
G13 $\quad poisoned_u \leftarrow \mathtt{F}$
G14 • $lx_u \leftarrow 0$
G15 • $\text{CH}_u \leftarrow \emptyset$
G16 • $xp_u \leftarrow \mathtt{F}$
G17 $(st_{\mathtt{A}}, st_{\mathtt{B}}) \leftarrow \text{init}$
G18 Invoke $\mathcal{A}$
G19 Lose

Oracle $\text{Send}(u, ad, m)$

S00 $c \leftarrow \text{send}\langle st_u\rangle(ad, m)$
S02 If is_u:
S03 $\quad \text{SC}_u \xleftarrow{\cup} \{(ad, c)\}$
S06 $lt_u \leftarrow lt_u + 1$
S07 Return c

Oracle $\text{Chal}(u, ad, m^0, m^1)$

C00 Require $m^0 \equiv m^1$
C01 Penalize $poisoned_u$
C02 If is_u: Penalize $xp_{\bar{u}}$
C03 $c \leftarrow \text{send}\langle st_u\rangle(ad, m^b)$
C04 If is_u:
C05 $\quad \text{SC}_u \xleftarrow{\cup} \{(ad, c)\}$
C06 $\quad$ If $is_{\bar{u}}$: $\text{CH}_u \xleftarrow{\cup} \{lt_u\}$
C07 $lt_u \leftarrow lt_u + 1$
C08 Return c

Oracle $\text{Expose}(u)$

• E00 Require $\text{CH}_{\bar{u}} = \emptyset$
E01 If is_u:
E02 $\quad \text{VF}_u[\![lt_u]\!] \xleftarrow{\cap} [\![lt_u]\!]$
E03 $\quad \text{AU}_u \xleftarrow{\cap} [\![lt_u]\!]$
• E04 $\quad lx_u \leftarrow lt_u$
• E05 $\quad xp_u \leftarrow \mathtt{T}$
E06 Return st_u

Oracle $\text{Decide}(b')$

D00 Stop with b'

Oracle $\text{Recv}(u, ad, c)$

R00 $(lt, pt) \leftarrow \text{ts}(c)$
R01 $(A, m) \leftarrow \text{recv}\langle st_u\rangle(ad, c)$
R06 If $(ad, c) \in \text{SC}_{\bar{u}}$:
R07 $\quad$ If is_u:
R09 $\quad\quad \text{AU}_{\bar{u}} \xleftarrow{\cup} \text{VF}_{\bar{u}}[lt]$
• R10 $\quad\quad$ If $lt \geq lx_{\bar{u}}$:
• R11 $\quad\quad\quad xp_{\bar{u}} \leftarrow \mathtt{F}$
• R15 $\quad$ If $lt \in \text{CH}_{\bar{u}}$:
• R16 $\quad\quad m \leftarrow \diamond$
R17 If $(ad, c) \notin \text{SC}_{\bar{u}}$:
R18 $\quad$ If is_u:
R19 $\quad\quad$ If $lt \notin \text{AU}_{\bar{u}}$:
R20 $\quad\quad\quad poisoned_u \leftarrow \mathtt{T}$
R23 $\quad is_u \leftarrow \mathtt{F}$
• R24 $\quad \text{CH}_{\bar{u}} \xleftarrow{\cap} [\![lt]\!]$
• R28 $\text{CH}_{\bar{u}} \leftarrow \text{CH}_{\bar{u}} \setminus \{lt\}$
R29 Return (A, m)

Oracle $\text{Tick}(u)$

T00 $\text{tick}\langle st_u\rangle$
T01 $pt_u \leftarrow pt_u + 1$
• T02 $\text{CH}_{\bar{u}} \xleftarrow{\cap} \text{ITC}(u)$

Fig. 2.3 Games $\text{CONF}^0, \text{CONF}^1$. Equivalence relation $\equiv$ on $\mathcal{M}$ [C00] is defined such that $m^0 \equiv m^1 :\Leftrightarrow |m^0| = |m^1|$. See text for the definition of function ITC [T02].

2.3 Games, Attacks and Examples

This section should be considered reference material to help to achieve a deeper understanding of Sec. 2.2. It includes examples for game executions and provides attack descriptions highlighting the subtleties of the security games defined in the previous section. It will not introduce new material and the reader may wish to skip ahead to Sec. 2.4.

2.3.1 Examples for BASIC game execution

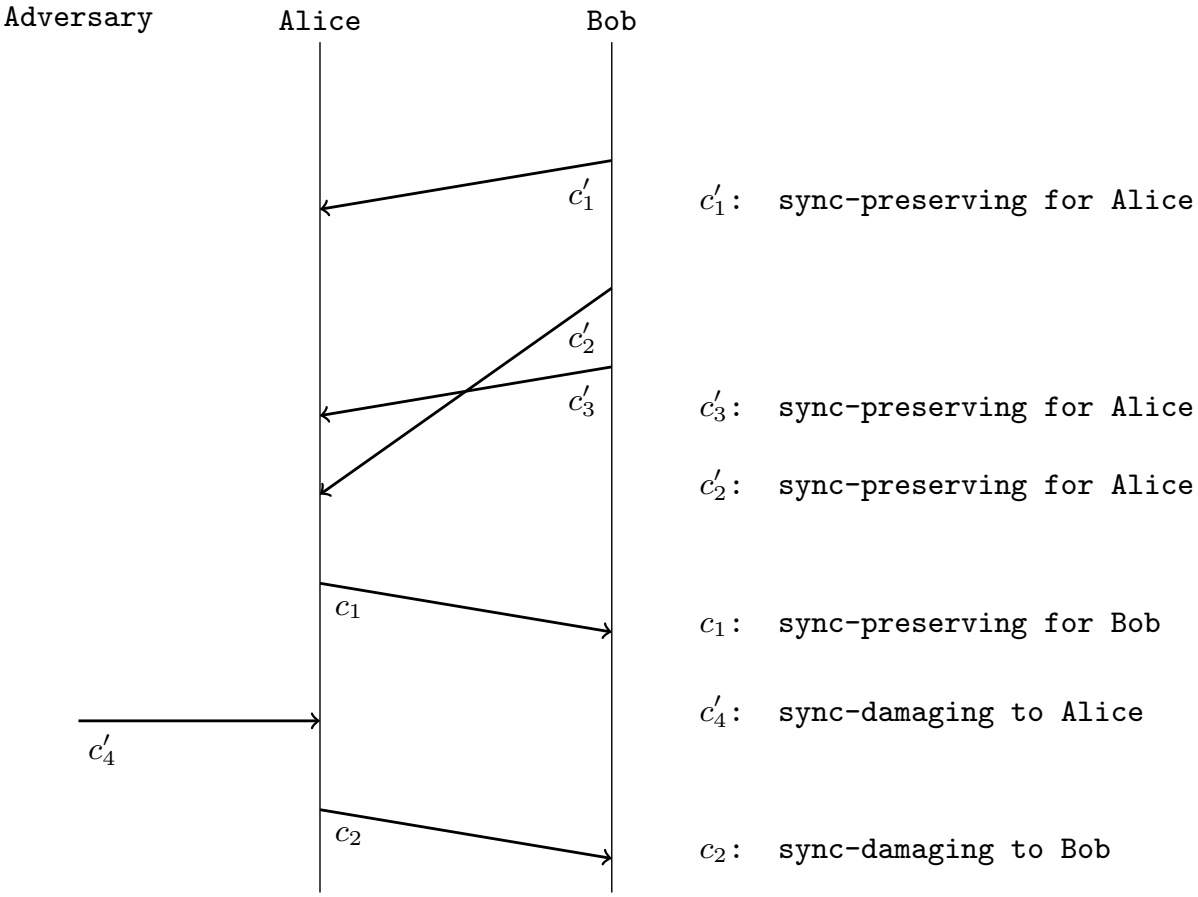

Fig. 2.4 Example of sync preserving and sync damaging ciphertexts. Time progresses from top to bottom in the diagram. The log reflects the events when *receiving* the specified ciphertext. Note c'_4 is injected by the adversary.

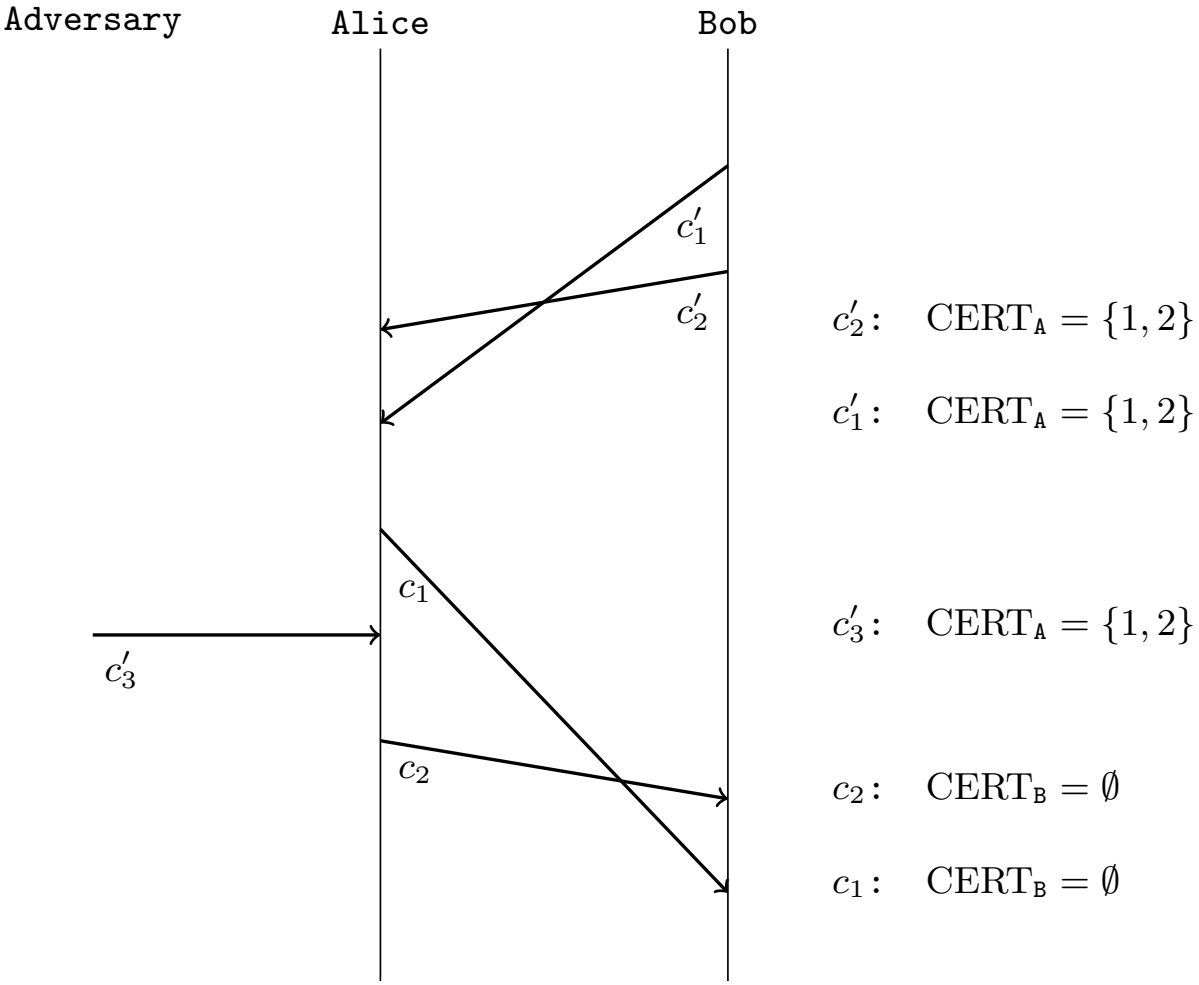

Fig. 2.5 Example of certifying ciphertexts. Time progresses from top to bottom in the diagram. The log reflects the relevant part of the game state when *receiving* the specified ciphertext. Observe that receiving c_1' does not alter $\mathrm{CERT}_{\mathtt{A}}$ because index 1 was already certified by c_2', and c_3' does not alter $\mathrm{CERT}_{\mathtt{A}}$ because it is injected by the adversary.

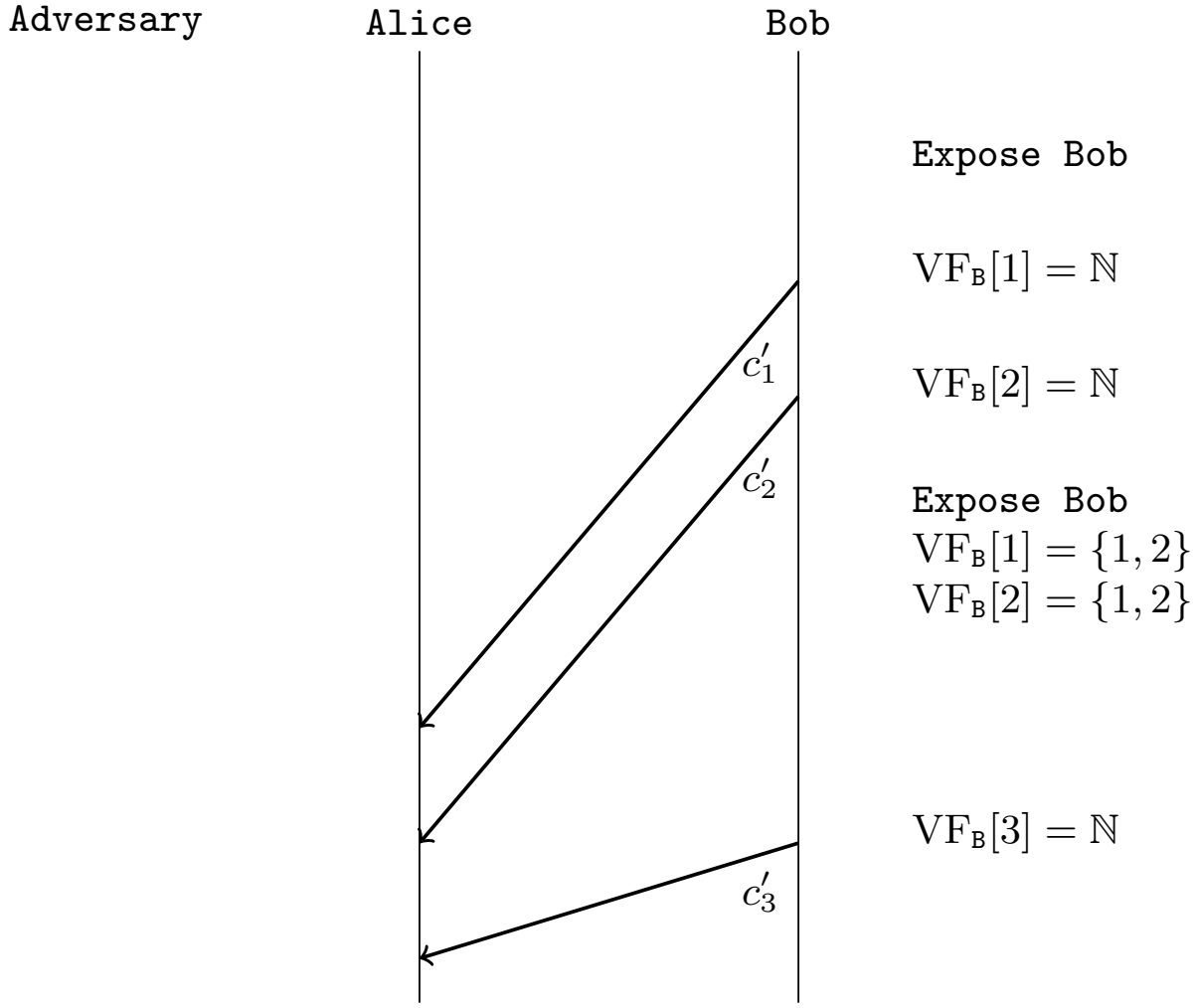

Fig. 2.6 Example of vouching ciphertexts. Time progresses from top to bottom in the diagram. Initially, each ciphertext c'_1, c'_2 vouches for its entire future (and past). This changes when the exposure of Bob happens after sending c'_2. Now c'_1 only vouches for its future up to the next exposure, i.e. only for c'_1 and c'_2. It should be clear that before any ciphertext is delivered the adversary can trivially forge any ciphertext. However, as soon as Alice receives c'_1, which vouches for c'_2, forging c'_2 is no longer a trivial task. Similarly, when Alice receives c'_3, which vouches for the entire future (and past), no forgeries would be trivial anymore (until the next exposure). The log reflects what a ciphertext vouches for and how this changes upon exposure.

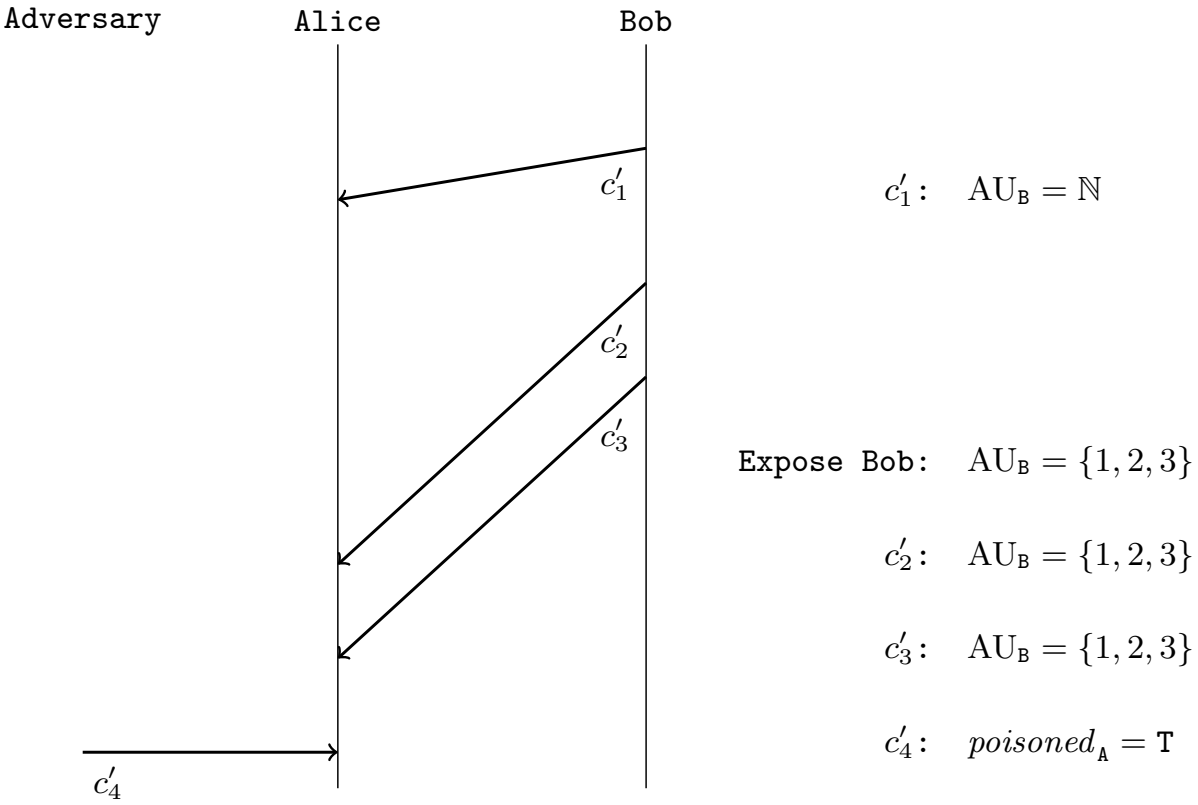

Fig. 2.7 Example of a poisoning ciphertext. Time progresses from top to bottom in the diagram. The log reflects the relevant part of the game state when *receiving* the specified ciphertext. Note c'_4 is injected by the adversary. This *poisons* Alice because at this point Alice is still in-sync and Bob is not authoritative for sending index 4.

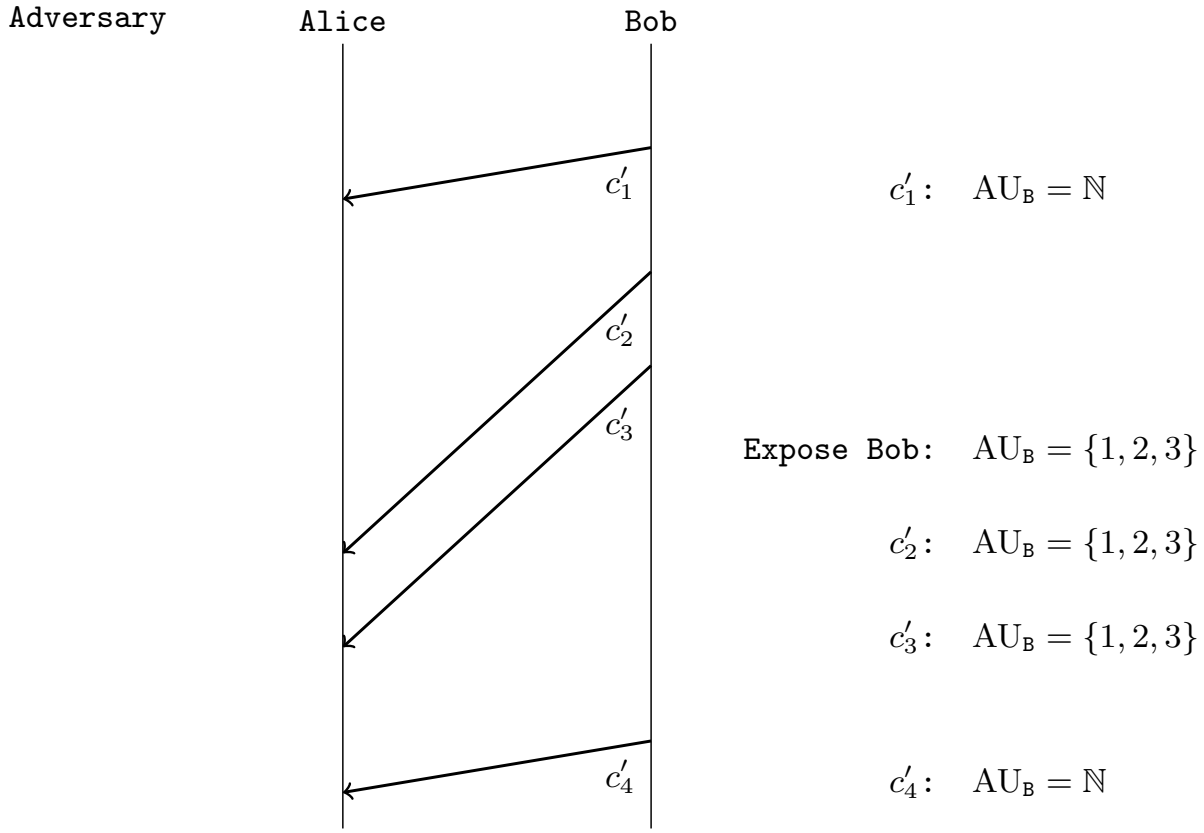

Fig. 2.8 Example of authoritative ciphertexts. Time progresses from top to bottom in the diagram. The log reflects the relevant part of the game state when *receiving* the specified ciphertext.

2.3.2 Attack Catalogue for Authenticity

The authenticity game AUTH in Fig. 2.2 excludes one trivial attack and it is exactly as one would intuitively expect:

1. Expose Alice and use her state to forge a message to Bob.
   $st_{\mathtt{A}} \leftarrow \text{Expose}(\mathtt{A})$; $c^* \leftarrow \text{send}\langle st_{\mathtt{A}}\rangle(m)$; $m^* \leftarrow \text{Recv}(\mathtt{B}, c^*)$.

The adversary is not rewarded for this trivial attack by [E03,R21].
The line numbers referenced in this subsection will refer to the lines in the AUTH game. For clarity we omit associated data as it is not relevant to the discussion. We now provide an overview of some attacks that are excluded in other work, but are *not* trivial and hence the AUTH game allows them. We remark that ACD [5] excludes all the attacks listed below.

I. Expose Alice and forge on the first message after the second is delivered.
   $st_{\mathtt{A}} \leftarrow \text{Expose}(\mathtt{A})$; $c_1^* \leftarrow \text{send}\langle st_{\mathtt{A}}\rangle(m_1)$; $c_1 \leftarrow \text{Send}(\mathtt{A}, m_1)$; $c_2 \leftarrow \text{Send}(\mathtt{A}, m_2)$; $m_2 \leftarrow \text{Recv}(\mathtt{B}, c_2)$; $m_1^* \leftarrow \text{Recv}(\mathtt{B}, c_1^*)$.

   The adversary is rewarded for this trivial attack by [R09,R21].

II. Expose Alice and forge on the second message after the first is delivered.
   $st_{\mathtt{A}} \leftarrow \text{Expose}(\mathtt{A})$; $c_1 \leftarrow \text{Send}(\mathtt{A}, m_1)$; $\_ \leftarrow \text{send}\langle st_{\mathtt{A}}\rangle(m_1)$; $c_2^* \leftarrow \text{send}\langle st_{\mathtt{A}}\rangle(m_2)$; $m_1 \leftarrow \text{Recv}(\mathtt{B}, c_1)$; $m_2^* \leftarrow \text{Recv}(\mathtt{B}, c_2^*)$.

   The adversary is rewarded for this trivial attack by [R09,R21].

III. Forge on an old message after bringing Bob out-of-sync with a trivial attack.
   $st_{\mathtt{A}} \leftarrow \text{Expose}(\mathtt{A})$; $c_1^* \leftarrow \text{send}\langle st_{\mathtt{A}}\rangle(m_1)$; $c_1 \leftarrow \text{Send}(\mathtt{A}, m_1)$; $c_2 \leftarrow \text{Send}(\mathtt{A}, m_2)$; $m_2 \leftarrow \text{Recv}(\mathtt{B}, c_2)$; $st_{\mathtt{A}} \leftarrow \text{Expose}(\mathtt{A})$; $c_3^* \leftarrow \text{send}\langle st_{\mathtt{A}}\rangle(m_3)$; $m_3^* \leftarrow \text{Recv}(\mathtt{B}, c_3^*)$; $m_1^* \leftarrow \text{Recv}(\mathtt{B}, c_1^*)$.

   The adversary is rewarded for this trivial attack by [R08,R22]. Note that the adversary wins the AUTH game by injecting a message to an out-of-sync Bob, which contrasts to the first attack where Bob was in-sync.

IV. Let an out-of-sync Bob send a message to Alice.
   $st_{\mathtt{A}} \leftarrow \text{Expose}(\mathtt{A})$; $c_1^* \leftarrow \text{send}\langle st_{\mathtt{A}}\rangle(m_1)$; $m_1^* \leftarrow \text{Recv}(\mathtt{B}, c_1^*)$; $c_1' \leftarrow \text{Send}(\mathtt{B}, m_1')$; $m_1' \leftarrow \text{Recv}(\mathtt{A}, c_1')$;

   The adversary is rewarded for this trivial attack by [S02,R21].

V. Expose an out-of-sync Bob to forge a message to Alice.

$st_{\mathtt{A}} \leftarrow \text{Expose}(\mathtt{A});\ c_1^* \leftarrow \text{send}\langle st_{\mathtt{A}}\rangle(m_1);\ m_1^* \leftarrow \text{Recv}(\mathtt{B}, c_1^*);\ st_{\mathtt{B}} \leftarrow \text{Expose}(\mathtt{B});\ c_1' \leftarrow \text{send}\langle \mathtt{B}\rangle(m_1');\ m_1' \leftarrow \text{Recv}(\mathtt{A}, c_1');$

The adversary is rewarded for this trivial attack by [E01,R21].

2.3.3 Attack Catalogue for Confidentiality

We provide an overview of trivial attacks that are prevented by the confidentiality game CONF in Fig. 2.3. The line numbers referenced in this subsection will refer to the lines in the CONF game. We also demonstrate some attacks that look similar, but that are *not* trivial and hence the CONF game allows them. For clarity we omit associated data as it is not relevant to the discussion.

1. Let Alice send a challenge, expose Bob and decrypt the challenge ciphertext.

   $c^* \leftarrow \text{Chal}(\mathtt{A}, m^0, m^1);\ st_{\mathtt{B}} \leftarrow \text{Expose}(\mathtt{B});\ m^* \leftarrow \text{recv}\langle st_{\mathtt{B}}\rangle(c^*);\ \text{Decide}(m^* = m^0\ \mathtt{?}\ 0 : 1).$

   This trivial attack is prevented by [C06,E00].

   The following attacks look similar, but they are *not* trivial. The CONF game thus shall not exclude them. The attacks explain why line C06 in Fig. 2.3 is conditioned on both is_u and $is_{\bar{u}}$. For the following attacks we use notation $\$/n$ in the Recv oracle to indicate a random ciphertext for index n. (Thus note in particular confidentiality can be maintained processing random ciphertexts, independent of authenticity.)

   I. First bring Bob out-of-sync for an index that is earlier than Alice's index, then expose him.

   $\_ \leftarrow \text{Send}(\mathtt{A}, m);\ \_ \leftarrow \text{Send}(\mathtt{A}, m);\ \_ \leftarrow \text{Recv}(\mathtt{B}, \$/0);\ c^* \leftarrow \text{Chal}(\mathtt{A}, m^0, m^1);\ st_{\mathtt{B}} \leftarrow \text{Expose}(\mathtt{B});\ m^* \leftarrow \text{recv}\langle st_{\mathtt{B}}\rangle(c^*);\ \text{Decide}(m^* = m^0\ \mathtt{?}\ 0 : 1).$

   II. First bring Bob out-of-sync for an index that is later than Alice's index, then expose him.

   $\_ \leftarrow \text{Recv}(\mathtt{B}, \$/5);\ c^* \leftarrow \text{Chal}(\mathtt{A}, m^0, m^1);\ st_{\mathtt{B}} \leftarrow \text{Expose}(\mathtt{B});\ m^* \leftarrow \text{recv}\langle st_{\mathtt{B}}\rangle(c^*);\ \text{Decide}(m^* = m^0\ \mathtt{?}\ 0 : 1).$

III. First bring Alice out-of-sync (without poisoning her, see below), then let her send a challenge.

$\_ \leftarrow \text{Recv}(\mathtt{A}, \$/0)$; $c^* \leftarrow \text{Chal}(\mathtt{A}, m^0, m^1)$; $st_{\mathtt{B}} \leftarrow \text{Expose}(\mathtt{B})$; $m^* \leftarrow \text{recv}\langle st_{\mathtt{B}}\rangle(c^*)$; $\text{Decide}(m^* = m^0 \ ? \ 0 : 1)$.

IV. It is also not a trivial attack if Bob's clock ticks $\delta + 1$ times before the exposure. The game allows this attack via [T02].

$c^* \leftarrow \text{Chal}(\mathtt{A}, m^0, m^1)$; $\text{Tick}(\mathtt{B})$; $\ldots$; $\text{Tick}(\mathtt{B})$; $st_B \leftarrow \text{Expose}(\mathtt{B})$; $m^* \leftarrow \text{recv}\langle st_{\mathtt{B}}\rangle(c^*)$; $\text{Decide}(m^* = m^0 \ ? \ 0 : 1)$.

2. Expose Alice, poison Bob, let Bob send a challenge:

$st_{\mathtt{A}} \leftarrow \text{Expose}(\mathtt{A})$; $c \leftarrow \text{send}\langle st_{\mathtt{A}}\rangle(m)$; $m' \leftarrow \text{Recv}(\mathtt{B}, c)$; $c^* \leftarrow \text{Chal}(\mathtt{B}, m^0, m^1)$; $m^* \leftarrow \text{recv}\langle st_{\mathtt{A}}\rangle(c^*)$; $\text{Decide}(m^* = m^0 \ ? \ 0 : 1)$.

This is prevented by [C01,R20].

3. Expose Bob, let Alice send a challenge:

$st_{\mathtt{B}} \leftarrow \text{Expose}(\mathtt{B})$; $c^* \leftarrow \text{Chal}(\mathtt{A}, m^0, m^1)$; $m^* \leftarrow \text{recv}\langle st_{\mathtt{B}}\rangle(c^*)$; $\text{Decide}(m^* = m^0 \ ? \ 0 : 1)$.

This is prevented by [C02,E05].

V. The attack is not trivial if Bob sends a message after the exposure that is delivered to Alice before the challenge:

$st_{\mathtt{B}} \leftarrow \text{Expose}(\mathtt{B})$; $c \leftarrow \text{Send}(\mathtt{B}, m)$; $m \leftarrow \text{Recv}(\mathtt{A}, c)$; $c^* \leftarrow \text{Chal}(\mathtt{A}, m^0, m^1)$; $m^* \leftarrow \text{recv}\langle st_{\mathtt{B}}\rangle(c^*)$; $\text{Decide}(m^* = m^0 \ ? \ 0 : 1)$.

This is allowed by [E04,E05,R10,R11].

VI. Observe that the attack is also not trivial if Bob was rendered out-of-sync before the exposure:

$st_{\mathtt{A}} \leftarrow \text{Expose}(\mathtt{A})$; $c_1 \leftarrow \text{send}\langle st_{\mathtt{A}}\rangle(m_1)$; $c_2 \leftarrow \text{send}\langle st_{\mathtt{A}}\rangle(m_2)$; $m_2 \leftarrow \text{Recv}(\mathtt{B}, c_2)$; $st_{\mathtt{B}} \leftarrow \text{Expose}(\mathtt{B})$; $c^* \leftarrow \text{Chal}(\mathtt{A}, m^0, m^1)$; $m^* \leftarrow \text{recv}\langle st_{\mathtt{B}}\rangle(c^*)$; $\text{Decide}(m^* = m^0 \ ? \ 0 : 1)$.

This is allowed by [E01].

4. Let Alice send a challenge, let Bob receive it:

$c^* \leftarrow \mathrm{Chal}(\mathtt{A}, m^0, m^1);\ m^* \leftarrow \mathrm{Recv}(\mathtt{B}, c^*);\ \mathrm{Decide}(m^* = m^0\ ?\ 0 : 1).$

This is prevented by [R16]. To see why this is not conditioned is_u we expand to the following trivial attack:
Let Alice send a challenge, expose Alice to trivially forge a ciphertext and let Bob receive it to go out-of-sync before receiving the challenge ciphertext:

$c^* \leftarrow \mathrm{Chal}(\mathtt{A}, m^0, m^1);\ \ st_{\mathtt{A}} \leftarrow \mathrm{Expose}(\mathtt{A});\ \ c_2 \leftarrow \mathrm{send}\langle st_{\mathtt{A}}\rangle(m);\ \ m_2 \leftarrow \mathrm{Recv}(\mathtt{B}, c_2);$
$m^* \leftarrow \mathrm{Recv}(\mathtt{B}, c^*);\ \mathrm{Decide}(m^* = m^0\ ?\ 0 : 1).$

VII. However, the following variation where the sending index of the challenge and the forgery is swapped, is *not* a trivial attack:

$st_{\mathtt{A}} \leftarrow \mathrm{Expose}(\mathtt{A});\ c_1 \leftarrow \mathrm{Send}(\mathtt{A}, m);\ c^* \leftarrow \mathrm{Chal}(\mathtt{A}, m^0, m^1);\ c_1' \leftarrow \mathrm{send}\langle st_{\mathtt{A}}\rangle(m');$
$m' \leftarrow \mathrm{Recv}(\mathtt{B}, c_1');\ m^* \leftarrow \mathrm{Recv}(\mathtt{B}, c^*);\ \mathrm{Decide}(m^* = m^0\ ?\ 0 : 1).$

The game will reveal the message m^* by [R15,R24].

5. Similarly to the previous item, there is a trivial attack when the final step is to expose an out-of-sync Bob instead of letting Bob receive the challenge ciphertext:

$c^* \leftarrow \mathrm{Chal}(\mathtt{A}, m^0, m^1);\ \ st_{\mathtt{A}} \leftarrow \mathrm{Expose}(\mathtt{A});\ \ c_2 \leftarrow \mathrm{send}\langle st_{\mathtt{A}}\rangle(m);\ \ m_2 \leftarrow \mathrm{Recv}(\mathtt{B}, c_2);$
$st_{\mathtt{B}} \leftarrow \mathrm{Expose}(\mathtt{B});\ m^* \leftarrow \mathrm{recv}\langle st_{\mathtt{B}}\rangle(c^*);\ \mathrm{Decide}(m^* = m^0\ ?\ 0 : 1).$

This is prevented by [E00,R24].

VIII. Again, it is not a trivial attack when the sending index of the forgery and challenge is swapped:

$st_{\mathtt{A}} \leftarrow \mathrm{Expose}(\mathtt{A});\ c_1 \leftarrow \mathrm{Send}(\mathtt{A}, m);\ c^* \leftarrow \mathrm{Chal}(\mathtt{A}, m^0, m^1);\ c_1' \leftarrow \mathrm{send}\langle st_{\mathtt{A}}\rangle(m');$
$m' \leftarrow \mathrm{Recv}(\mathtt{B}, c_1');\ st_{\mathtt{B}} \leftarrow \mathrm{Expose}(\mathtt{B});\ m^* \leftarrow \mathrm{recv}\langle st_{\mathtt{B}}\rangle(c^*);\ \mathrm{Decide}(m^* = m^0\ ?\ 0 :$
$1).$

This is allowed by [E00,R24].

2.3.4 Attack on Forward Secrecy of ACD [5]

It is straightforward to see the protocol of ACD does not satisfy our notion of forward secrecy. Consider an adversary that challenges Alice, obtaining a ciphertext $c \leftarrow \text{Chal}(\mathtt{A}, ad, m^0, m^1)$, but never delivers the ciphertext to Bob. Now, if time passes, by repeated Tick invocations, the adversary is allowed to Expose($\mathtt{B}$) and obtain Bob's user state. The ACD protocol has no concept of time and the state will still be able to decrypt the ciphertext.

Irrespective of physical time, further observe that the ACD protocol allows forgeries for 'logical' old ciphertext. Consider both parties' states public, that is the adversary exposes them (before and after) each sending invocation. Now the adversary remains passive and makes the following oracle queries: $c_1 \leftarrow \text{Send}(\mathtt{A}, ad, m)$, $c_2 \leftarrow \text{Send}(\mathtt{A}, ad, m)$ and $\text{Recv}(\mathtt{B}, ad, c_2)$. Our authenticity notion demands the adversary is unable to create a forgery for the first ciphertext. However, in the ACD protocol ciphertexts do not 'pin' earlier ciphertexts and the adversary is able to forge the first ciphertext for any arbitrary message.

2.4 Non-interactive Primitives

In Sec. 2.2 we defined the syntax and security of KuBOOM and we will provide a secure construction in Sec. 2.5. The current section is dedicated to presenting a set of cryptographic building blocks, in the spirit of public key encryption (PKE) and signature schemes (SS), that will play crucial roles in our construction. Recall that a defining property of a KuBOOM protocol is that it provides maximum resilience against (continued) state exposure attacks, i.e. preventing all but trivial attacks. If a construction would rely on regular PKE or SS schemes as building blocks, the secret keys of the latter would leak on state exposure, which in most cases would inevitably clear the way for an attack on confidentiality or authenticity. It is thus a common engineering principle in the secure messaging domain to employ variants of PKE and SS that are stateful (instead of statically keyed) and process their internal keying material after each use to an updated 'refreshed' version that limits the options of a state exposing adversary to harm only future operations (while past operations remain protected).[19] In fact, some of the building blocks that we propose here additionally fold an auxiliary external input into their state when updating the latter. We refer to this auxiliary input as the *associated*

[19]While the PKE and SS variants that we consider in the current section specifically provide 'forward secrecy', we do not require them to offer 'post-compromise security' as well, i.e., we don't expect them to auto-heal.

data (of the update), and the assumption is that sender and receiver (i.e., signer and verifier, or encryptor and decryptor) update their states with consistent such inputs (independently of each other).[20]

We note that while the specifics of our building blocks might be different from those of prior work, it can be generally considered well-understood how to construct such primitives. For instance, a forward-secure SS [19], which is a primitive close to one of ours, can be built by coupling each signing operation with the generation of a fresh signature key pair, the public component of which is signed and thus authenticated along with the message; after the signing operation is complete, the original signing key is disposed of and replaced by the freshly generated one. Adding the support of auxiliary associated-data strings into such a scheme is trivial (just authenticate the string along with the message) and is less a cryptographic challenge than an exercise of maintaining the right data structures in the sender/receiver state. Similarly, forward-secure PKE [36], which is a primitive close to one of ours as well, is routinely built from hierarchical identity-based encryption (HIBE) by associating key validity epochs with the nodes of a binary tree. Variants of forward-secure PKE that support key updates that depend on auxiliary associated-data strings have been proposed in prior work as well [56, 86], using design approaches that can be seen as minor variations of the original tree-based idea from [36].

For our KuBOOM construction in Sec. 2.5 we require three independent forward-secure public key primitives which we refer to as *updatable signature scheme*, *key-updatable KEM*, and *key-evolving KEM*, respectively. We specify their syntax and the expected behaviour below, formalize the security definitions and propose concrete constructions. We note, however, that our security definitions and constructions can be seen as following immediately from the syntax and expected functionality: While the security definitions give the adversary the option to expose the state of any participant any number of times, and formalize the best-possible security that is feasible under such a regime (i.e., maximum resilience against state exposure attacks), the constructions, which all follow the approaches of [19, 36, 56, 86] discussed above, are engineered to re-generate fresh key material whenever an opportunity for this arises. We show that these strategies (to formalize security and to develop a construction) match well together: Our constructions

[20] Unlike regular signature schemes where for each signer there can be many independent verifiers, and unlike regular public key encryption where for each decryptor there can be many encryptors, for the primitives we consider in the current section a strict one-to-one correspondence between sender and receiver is assumed. For instance, in the PKE setting, if sender Alice updates her (public) key with associated-data string `ABC` while receiver Bob updates his (secret) key with associated-data string `XYZ`, then the ciphertexts that Alice generates cannot be decrypted by Bob.

are secure in our models (and thus, in a sense, optimal). In the remaining part of this section, we analyse the three mentioned building blocks.

2.4.1 Updatable Signature Schemes (USS)

An *updatable signature scheme* (USS) assumes two parties, a signer and a (single) verifier, and lets the former generate signatures on messages that the latter can then check for authenticity. Like a regular signature scheme (Sec. 1.5.2), a USS has algorithms $\mathsf{gen}, \mathsf{sign}, \mathsf{vfy}$, where sign creates a signature on a given message and vfy verifies that a given signature is valid for a given message. The particularity of USS is that signing and verification keys can be updated with external information, and that signatures only verify correctly if these updates are performed consistently. Crucially, both parties maintain states which can be *updated* by feeding them with arbitrary input strings that we refer to as *associated data*. More precisely, signing and verification keys are replaced by signing and verification *states*, and update algorithms $\mathsf{updss}, \mathsf{updvs}$ are added (for 'update signing state' and 'update verification state', respectively) that take an associated-data string and fold it cryptographically into the respective state. Multiple such update operations can be performed in succession, both for the signing keys and verification keys. If the states of the signer and the verifier are updated inconsistently, i.e., with different (sequences of) associated-data strings, signatures created by the signer shall not be deemed valid by the verifier. Below we first specify the syntax and functionality of USS. We then propose a strong authenticity notion that demands that a maximum level of unforgeability is provided in a setting where the adversary can reveal the states of both parties.

We remark a similar primitive is considered in [56], but their construction and security requirement is different. They use a *forward secure* or *evolving* signature scheme (introduced in [19]), meaning it can only update with an empty *ad* string, as building block. The construction in [56] is effectively a certification chain: it signs the associated-data and then 'evolves' the signing key. The resulting signature is a sequence of the produced signatures that have to be sequentially checked by the verifier. However, as [19] correctly point out, if one is willing to accept certification chains the solution is straightforward: At each update, the signer generates a new key pair and signs the associated-data and the new verification key with the old signing key, which is subsequently deleted. Thus we do not require the evolving signature primitive, instead we use (one-time) signatures. Finally, [56] requires unique updatable signatures for their messaging construction, but we note that we do not need this property.

Example 1	**Example 2**	**Example 3**
X00 $(ss, vs) \leftarrow \text{gen}$	X10 $(ss, vs) \leftarrow \text{gen}$	X20 $(ss, vs) \leftarrow \text{gen}$
X01 For $ad \leftarrow$ "A" to "Z":	X11 $\text{updss}\langle ss\rangle(\text{"A"})$	X21 $\text{updss}\langle ss\rangle(\text{"A"})$
X02 $\quad \text{updss}\langle ss\rangle(ad)$	X12 $\text{updss}\langle ss\rangle(\text{"B"})$	X22 $\sigma \leftarrow \text{sign}\langle ss\rangle(\text{"MSG"})$
X03 $\sigma \leftarrow \text{sign}\langle ss\rangle(\text{"MSG"})$	X13 $\sigma \leftarrow \text{sign}\langle ss\rangle(\text{"MSG"})$	X23 $\text{updvs}\langle vs\rangle(\text{"A"})$
X04 For $ad \leftarrow$ "A" to "Z":	X14 $\text{updvs}\langle vs\rangle(\text{"B"})$	X24 $\text{updvs}\langle vs\rangle(\text{"A"})$
X05 $\quad \text{updvs}\langle vs\rangle(ad)$	X15 $\text{updvs}\langle vs\rangle(\text{"A"})$	X25 $\text{vfy}(vs, \text{"MSG"}, \sigma)$ ✗
X06 $\text{vfy}(vs, \text{"MSG"}, \sigma)$ ✓	X16 $\text{vfy}(vs, \text{"MSG"}, \sigma)$ ✗	

Fig. 2.9 Examples of USS use.

Syntax. A *key-updatable signature* scheme for a message space $\mathcal{M}$ and an associated-data space $\mathcal{AD}$ consists of a signing state space $\mathcal{SS}$, a verification state space $\mathcal{VS}$, a signature space Σ, and algorithms $\text{gen}, \text{sign}, \text{vfy}, \text{updss}, \text{updvs}$ with APIs

$$\text{gen} \rightarrow \mathcal{SS} \times \mathcal{VS} \qquad \mathcal{M} \rightarrow \text{sign}\langle \mathcal{SS} \rangle \rightarrow \Sigma \qquad \mathcal{VS} \times \mathcal{M} \times \Sigma \rightarrow \text{vfy}$$

and

$$\mathcal{AD} \rightarrow \text{updss}\langle \mathcal{SS} \rangle \qquad \mathcal{AD} \rightarrow \text{updvs}\langle \mathcal{VS} \rangle \ .$$

Note that the vfy algorithm does not have an explicit output. We say that algorithm vfy accepts if it terminates normally; otherwise, if it aborts (with an error code), we say it rejects. We expect of a correct USS that for all $(ss, vs) \in [\text{gen}]$, if ss and vs are updated by invoking $\text{updss}\langle ss\rangle(\cdot)$ and $\text{updvs}\langle vs\rangle(\cdot)$ with the same sequence $ad_1, \ldots, ad_l \in \mathcal{AD}$ of associated data, then for all $m \in \mathcal{M}$ and $\sigma \in [\text{sign}\langle ss\rangle(m)]$ we have that $\text{vfy}(vs, m, \sigma)$ accepts.

USS Examples. In Fig. 2.9 we illustrate the expected behaviour of a USS with three examples. When the three blocks of code are executed, we expect the vfy invocation of [X06] to accept, and those of [X16,X25] to reject.

Unforgeability under State Exposure. We require of a USS that it provides strong unforgeability, with the strongest possible model for state corruption. This is formalized via Game AUTH in Fig. 2.10, where oracles Updss, Updvs, Sign, Vfy give access to the corresponding scheme algorithms, and oracles ExposeS, ExposeV reveal user states. Variables AD_S, AD_V ('Associated Data', [G00]) store the associated-data strings that the signer and the verifier, respectively, provided so far [U01,U11]. Set SM records information associated with the signing queries: The signer's update strings so far, the signed message, and the generated signature [G01,S01,S02]. Set XP indicates for which update strings the signer has been exposed, i.e., shares the ability to generate valid

Game AUTH($\mathcal{A}$)
G00 $\mathrm{AD}_S, \mathrm{AD}_V \leftarrow ()$
G01 $\mathrm{SM} \leftarrow \emptyset$
G02 $\mathrm{XP} \leftarrow \emptyset$
G03 $(ss, vs) \leftarrow \mathrm{gen}$
G04 Invoke $\mathcal{A}$
G05 Lose

Oracle Updss(ad)
U00 $\mathrm{updss}\langle ss\rangle(ad)$
U01 $\mathrm{AD}_S \xleftarrow{\#} ad$

Oracle Sign(m)
S00 $\sigma \leftarrow \mathrm{sign}\langle ss\rangle(m)$
S01 $cid := (\mathrm{AD}_S, m, \sigma)$
S02 $\mathrm{SM} \xleftarrow{\cup} \{cid\}$
S03 Return σ

Oracle ExposeS
E00 $\mathrm{FE}_S := \{\mathrm{AD} : \mathrm{AD}_S \preceq \mathrm{AD}\}$
E01 $\mathrm{XP} \xleftarrow{\cup} \mathrm{FE}_S$
E02 Return ss

Oracle Updvs(ad)
U10 $\mathrm{updvs}\langle vs\rangle(ad)$
U11 $\mathrm{AD}_V \xleftarrow{\#} ad$

Oracle Vfy(m, σ)
V00 $\mathrm{vfy}(vs, m, \sigma)$
V01 $cid := (\mathrm{AD}_V, m, \sigma)$
V02 If $cid \notin \mathrm{SM}$:
V03 Reward $\mathrm{AD}_V \notin \mathrm{XP}$

Oracle ExposeV
E10 Return vs

Fig. 2.10 Game AUTH for USS.

signatures with the adversary: Initially this set is empty [G02], but when the signer's state is exposed, all update strings which have AD_S as prefix are added to the set [E00,E01]. The game encodes one winning condition: The adversary is rewarded for presenting a signature that is not a replay if they had not exposed the signer's state for the verifier's update string [V02,V03]. A scheme under consideration provides unforgeability if the advantage $\mathbf{Adv}^{\mathrm{auth}}(\mathcal{A}) := \mathbb{P}[\mathrm{AUTH}(\mathcal{A})]$ is negligibly small for all realistic adversaries $\mathcal{A}$.

Construction. We provide a USS construction from (regular) signatures. The idea is that each update operation of the signer is implemented by (1) creating a fresh signature key pair, (2) certifying the new verification key together with the associated-data input with the old signing key, (3) securely overwriting the old signing key with the new signing key. Notably, as our application of USS in Sec. 2.5 requires only a single signature to be issued per signer instance, it suffices to instantiate the signature scheme with a (potentially more efficient or secure) one-time signature scheme.

Hence, in Fig. 2.11 we construct a USS, using a one-time signature scheme (OTS.gen, OTS.sign, OTS.v as a building block. The notation in [s02] means that the signing key embedded in state ss shall be securely deleted (by overwriting it with some value $\perp \notin \mathcal{SK}$). Further note that, after the first signing query, any subsequent queries will fail by line [s00].

We remark, even instantiated with one-time signatures, our construction will provide unforgeability as defined via Game AUTH in Fig. 2.10. To see this, observe the sign algorithm is stateful and can delete its signing key after the first signature. If an adversary

Proc gen
g00 $(sk, vk) \leftarrow \text{OTS.gen}$
g01 $A_S \leftarrow ()$
g02 $A_V \leftarrow ()$
g03 $ss := (sk, A_S)$
g04 $vs := (vk, A_V)$
g05 Return (ss, vs)

Proc updss$\langle ss \rangle(ad)$
u00 Require $sk \neq \bot$
u01 $(sk', vk') \leftarrow \text{OTS.gen}$
u02 $\sigma' \leftarrow \text{OTS.sign}(sk, \mathtt{U} + vk' + ad)$
u03 $sk \leftarrow sk'$
u04 $A_S \xleftarrow{+} (vk', \sigma')$

Proc updvs$\langle vs \rangle(ad)$
u10 $A_V \xleftarrow{+} ad$

Proc sign$\langle ss \rangle(m)$
s00 Require $sk \neq \bot$
s01 $\sigma' \leftarrow \text{OTS.sign}(sk, \mathtt{S} + m)$
s02 $sk \leftarrow \bot$
s03 $\sigma \leftarrow A_S + \sigma'$
s04 Return σ

Proc vfy(vs, m, σ)
v00 $A + \sigma' \leftarrow \sigma$
v01 $vk^* \leftarrow vk$; $A^* \leftarrow A_V$
v02 Require $|A| = |A^*|$
v03 While $A \neq () \wedge A^* \neq ()$:
v04 $\quad (vk', \sigma') + A \leftarrow A$
v05 $\quad ad + A^* \leftarrow A^*$
v06 $\quad \text{OTS.vfy}(vk^*, \mathtt{U} + vk' + ad, \sigma')$
v07 $\quad vk^* \leftarrow vk'$
v08 $\text{OTS.vfy}(vk^*, \mathtt{S} + m)$

Fig. 2.11 USS construction. In [u02,s01,v06,v08] we use the symbols $\mathtt{U}, \mathtt{S}$ to domain-separate updating and signing steps. 'Require C' is short for 'If $\neg C$: Abort'.

tries to query the Sign oracle in the AUTH game a second time, the sign algorithm will abort and not produce any output.

We note that our construction can be seen as a twist of the very similar construction described in [19]: instead of only signing the new verification key, we sign the verification key together with the associated-data string. Indeed, this construction has also been identified in recent work [66] and the security argument for our construction follows exactly in the footsteps of the arguments made in this work.

2.4.2 Key-updatable KEMs (KuKEM)

A *key-updatable key encapsulation mechanism* (KuKEM) is a stateful KEM variant with algorithms gen, enc, dec and the update properties of USS: both the encapsulator and the decapsulator can update their public/secret state material with algorithms updps, updss (for 'update public state' and 'update secret state', respectively) that take an associated-data string on input. The decapsulator, if updated in-sync with the encapsulator, can successfully decapsulate ciphertexts. As above, our security model formalizes IND-CCA-like security in a model supporting exposing the state of both parties, with the explicit requirement that state exposures do not harm the confidentiality of keys encapsulated for past epochs, nor the confidentiality of keys encapsulated with diverged states.

Game FUNC($\mathcal{A}$)
G00 $\mathrm{AD}_S, \mathrm{AD}_R \leftarrow ()$
G01 $\mathrm{SC} \leftarrow \emptyset$
G02 $\mathrm{K}[\cdot] \leftarrow \cdot$
G03 $(ss, ps) \leftarrow \mathrm{gen}$
G04 Invoke $\mathcal{A}$
G05 Lose

Oracle Enc
E00 $(k, c) \leftarrow \mathrm{enc}(ps)$
E01 $cid := (\mathrm{AD}_S, c)$
E02 Promise $cid \notin \mathrm{SC}$
E03 $\mathrm{SC} \xleftarrow{\cup} \{cid\}$
E04 $\mathrm{K}[cid] \leftarrow k$
E05 Return (k, c)

Oracle Dec(c)
D00 $k \leftarrow \mathrm{dec}(ss, c)$
D01 $cid := (\mathrm{AD}_R, c)$
D02 If $cid \in \mathrm{SC}$:
D03 Promise $k = \mathrm{K}[cid]$
D04 $k \leftarrow \diamond$
D05 Return k

Oracle Updps(ad)
U00 $\mathrm{updps}\langle ps\rangle(ad)$
U01 $\mathrm{AD}_S \xleftarrow{\#} ad$

Oracle Updss(ad)
U10 $\mathrm{updss}\langle ss\rangle(ad)$
U11 $\mathrm{AD}_R \xleftarrow{\#} ad$

Fig. 2.12 Game FUNC for KuKEM.

Syntax. A *key-updatable key encapsulation mechanism* for a key space $\mathcal{K}$ and an associated-data space $\mathcal{AD}$, consists of a secret state space $\mathcal{SS}$, a public state space $\mathcal{PS}$, a ciphertext space $\mathcal{C}$, KEM algorithms gen, enc, dec and state update algorithms updps, updss with APIs

$$\mathrm{gen} \to \mathcal{SS} \times \mathcal{PS} \qquad \mathcal{PS} \to \mathrm{enc} \to \mathcal{K} \times \mathcal{C} \qquad \mathcal{SS} \times \mathcal{C} \to \mathrm{dec} \to \mathcal{K}$$

$$\mathcal{AD} \to \mathrm{updps}\langle \mathcal{PS} \rangle \qquad \mathcal{AD} \to \mathrm{updss}\langle \mathcal{SS} \rangle \ .$$

We specify the expected functionality of a KuKEM in the FUNC game depicted in Fig. 2.12, where oracles Enc, Dec, Updps, Updss give access to the corresponding scheme algorithms. Epochs in the game are identified by an associated-data string AD [U01,U11], similar to the modelling for USS. Set SC records information associated with the encapsulation queries: the epoch and the generated ciphertext [G01,E01,E03] and the adversary is rewarded if ciphertexts are not unique within their epoch [E02]. Array K stores the key generated by the encapsulation algorithm [E04] and the adversary is rewarded if the decapsulation algorithm outputs a different key [D03]. We say that a KuKEM is *functional* if the probability $\mathbb{P}[\mathrm{FUNC}(\mathcal{A})]$ is negligibly small for all realistic adversaries $\mathcal{A}$.

Security of kuKEM. We require forward secure indistinguishability of the KuKEM scheme, which we formalize in models supporting user corruptions via the real-or-random style games defined in Fig. 2.13, where oracles Challenge, Dec, Updps, Updss give access to the corresponding scheme algorithms (with oracle Challenge giving left-or-right access to algorithm enc), oracles ExposeS, ExposeR reveal user states, and oracle Decide serves the adversary for delivering a guess on challenge bit b. Intuitively, the CONF games

Game $\text{CONF}^b(\mathcal{A})$

G00 $\text{AD}_S, \text{AD}_R \leftarrow ()$
G01 $\text{SC} \leftarrow \emptyset$
G02 $\text{CH} \leftarrow \emptyset$
G03 $\text{XP} \leftarrow \emptyset$
G04 $(ss, ps) \leftarrow \text{gen}$
G05 Invoke $\mathcal{A}$
G06 Lose

Oracle $\text{Decide}(b')$

D00 Stop with b'

Oracle Challenge

C00 Require $\text{AD}_S \notin \text{XP}$
C01 $(k^0, c) \leftarrow \text{enc}(ps)$
C02 $k^1 \leftarrow_\$ \mathcal{K}$
C03 $cid \coloneqq (\text{AD}_S, c)$
C04 $\text{SC} \xleftarrow{\cup} \{cid\}$
C05 $\text{CH} \xleftarrow{\cup} \{\text{AD}_S\}$
C06 Return (k^b, c)

Oracle $\text{Updps}(ad)$

U00 $\text{updps}\langle ps\rangle(ad)$
U01 $\text{AD}_S \xleftarrow{\#} ad$

Oracle ExposeS

E00 Return ps

Oracle $\text{Dec}(c)$

D10 $k \leftarrow \text{dec}(ss, c)$
D11 $cid \coloneqq (\text{AD}_R, c)$
D12 If $cid \in \text{SC}$:
D13 $\quad k \leftarrow \diamond$
D14 Return k

Oracle $\text{Updss}(ad)$

U10 $\text{updss}\langle ss\rangle(ad)$
U11 $\text{AD}_R \xleftarrow{\#} ad$

Oracle ExposeR

E10 $\text{FE} \coloneqq \{\text{AD} : \text{AD}_R \preceq \text{AD}\}$
E11 Require $\text{CH} \cap \text{FE} = \emptyset$
E12 $\text{XP} \xleftarrow{\cup} \text{FE}$
E13 Return ss

Fig. 2.13 Games $\text{CONF}^0, \text{CONF}^1$ for KuKEM.

require that using a secret state for decapsulation only produces the correct key if all secret state updates were consistent with the public state updates.

Set XP indicates for which epochs the receiver has been exposed: Initially this set is empty [G03], but when the receiver is exposed, all future epochs ('FE') are added to the set [E12]. The set FE ('future epochs')[21]: is the set of all associated-data strings AD which have the decapsulator's update string AD_R as prefix [E10]. Set CH tracks for which epochs the adversary has issued a challenge. Initially this set is empty [G02], but an epoch is added to the set in a Challenge query [C05]. The adversary is not allowed to challenge an exposed epoch [C00] and similarly not allowed to expose an active or future challenged epoch [E11]. The advantage of $\mathcal{A}$ is defined as $\mathbf{Adv}^{\text{CONF}}(\mathcal{A}) = |\mathbb{P}[\text{CONF}^1(\mathcal{A})] - \mathbb{P}[\text{CONF}^0(\mathcal{A})]|$. We say the scheme provides *indistinguishability* if the advantage $\mathbf{Adv}^{\text{CONF}}(\mathcal{A})$ that can be obtained is negligible for all realistic adversaries $\mathcal{A}$.

Construction of KuKEM. As we already alluded to earlier, a KuKEM can be built generically from hierarchical identity-based encryption (HIBE, [52]). We further note Lewko and Waters [73] provide an 'unbounded' HIBE in the sense that a maximum hierarchy depth does not have to be fixed during setup. In essence each hierarchy level is an instance of the Boneh-Boyen IBE scheme [24] with added randomness to employ a secret-sharing approach of the master secret key. These instances all share the same public parameters. We do not claim novelty of our KuKEM construction from HIBE and note that similar constructions can already be found in the literature [56, 86].

[21] Technically speaking FE also includes the currently active epoch.

Proc gen	**Proc** enc(ps)	**Proc** dec(ss, c)
g00 $(sk, pk) \leftarrow \text{Hgen}$	e00 $(k, c) \leftarrow \text{Henc}(pk, \text{AD}_S)$	d00 $k \leftarrow \text{Hdec}(sk, c)$
g01 $ss := sk$	e01 Return (k, c)	d01 Return k
g02 $\text{AD}_S \leftarrow ()$	**Proc** updps$\langle ps \rangle(ad)$	**Proc** updss$\langle ss \rangle(ad)$
g03 $ps := (pk, \text{AD}_S)$	u00 $\text{AD}_S \xleftarrow{\#} ad$	u10 $sk \leftarrow \text{Hdelegate}(sk, ad)$
g04 Return (ss, ps)		

Fig. 2.14 KuKEM construction. Building block is a HIBE-KEM (Hgen, Henc, Hdec, Hdelegate).

We will sketch the general idea of the construction and refer the reader to Fig. 2.14 for a detailed description. Essentially, a KuKEM is a descending chain in the hierarchy, where the current level can be used for encapsulation and decapsulation. In more detail, the gen procedure runs the Hgen procedure to generate a secret and public key pair (sk, pk). The string AD_S is initialized empty. The private and public information is stored in the respective states. The encapsulation procedure enc encapsulates to the identity AD_S. The updps procedure updates the public state with associated data ad by embedding ad into AD_S. To update the secret state, updss delegates the current secret key sk to identity ad below it in the hierarchy. The decapsulation procedure dec uses the secret key stored in the secret state to decapsulate the ciphertext.

We remark we do not need the full capabilities of an HIBE scheme that can arbitrarily branch off: we only require one descending chain. A more direct construction of a KuKEM is currently still an open problem.

2.4.3 Key-evolving KEMs (KeKEM)

A *key-evolving key encapsulation mechanism* (KeKEM) consists of algorithms gen, enc, dec like a regular KEM, but, as above, public and secret keys are replaced by public and secret states, respectively, that can be updated. More precisely, the encapsulator's and decapsulator's states can be updated 'to the next epoch' by invoking the evolveps (for 'evolve public state') algorithm and the evolvess (for 'evolve secret state') algorithm, respectively. Note, however, that if a secret state is updated the decryptability of ciphertexts generated for older epochs is not automatically lost; rather, ciphertexts associated to multiple epochs remain decryptable until epochs are explicitly declared redundant by invoking the expire algorithm.[22] A somewhat related primitive was considered in [36]. Our security model formalizes IND-CCA-like security in a model

[22]The expire algorithm expires always the oldest currently supported epoch. That is, active epochs of KeKEMs always span a continuous interval.

supporting exposing the state of both parties, with the explicit requirement that state exposures do not harm the confidentiality of keys encapsulated for expired epochs. Note that our formalization of KeKEMs does not support updating states with respect to an associated-data input.

Syntax. A *key-evolving key encapsulation mechanism* for a key space $\mathcal{K}$ consists of a secret state space $\mathcal{SS}$, a public state space $\mathcal{PS}$, a ciphertext space $\mathcal{C}$, KEM algorithms $\mathsf{gen}, \mathsf{enc}, \mathsf{dec}$ and state update algorithms $\mathsf{evolveps}, \mathsf{evolvess}, \mathsf{expire}$ with APIs

$$\mathbb{N} \to \mathsf{gen} \to \mathcal{SS} \times \mathcal{PS} \qquad \mathcal{PS} \to \mathsf{enc} \to \mathcal{K} \times \mathcal{C} \qquad \mathcal{SS} \times \mathbb{N} \times \mathcal{C} \to \mathsf{dec} \to \mathcal{K}$$

and

$$\mathsf{evolveps}\langle \mathcal{PS} \rangle \qquad \mathsf{evolvess}\langle \mathcal{SS} \rangle \qquad \mathsf{expire}\langle \mathcal{SS} \rangle \ .$$

In the KeKEM setting it makes sense to number the epochs. Note that the dec algorithm expects, besides the secret state and the ciphertext, an explicit indication of the epoch number for which the ciphertext was created. For simplicity, one would like to provide an absolute time to the dec algorithm, e.g. unix time, rather than the time offset relative to the generation time. For this reason, the gen algorithm takes in an epoch number which can be used to specify the generation time and thus the first epoch need not necessarily start at zero. Then the state can internally compute the relative offset on decapsulation. As the full definition is quite involved, we first illustrate the functionality of a correct KeKEM using an example: If we invoke $(ss, ps) \leftarrow \mathsf{gen}(5)$ to generate a state pair (and associating the number 5 with the first state), then invoking 2-times $\mathsf{evolveps}\langle ps \rangle$ followed by $(k, c) \leftarrow \mathsf{enc}(ps)$, and then 4-times $\mathsf{evolvess}\langle ss \rangle$, then invoking $\mathsf{dec}(ss, 7, c)$ will return k until $\mathsf{expire}\langle ss \rangle$ has been invoked for the third time (expiring epochs 5, 6, and finally 7).

We specify the expected functionality of a KeKEM in the FUNC game depicted in Fig. 2.15, where oracles Enc, Dec, Evolvess, Evolveps and Expire give access to the corresponding scheme algorithms. Set AE ('Active Epochs', [G02]) stores the span of epochs that the receiver has opened up [E22] but not yet expired [X02]. Observe that $\mathrm{AE} = \{ft_R, \ldots, pt_R\}$. Set SC records information associated with the encapsulation queries: the epoch and the generated ciphertext [G03,E01,E03] and the adversary is rewarded if ciphertexts are not unique within their epoch [E02]. Array K stores the key generated by the encapsulation algorithm [E04] and the adversary is rewarded if the decapsulation algorithm outputs a different key [D04]. Note that the decapsulation algorithm is also expected to abort on an epoch that is not active [D01]. We say that

Game FUNC$(pt, \mathcal{A})$
G00 $pt_S \leftarrow pt$
G01 $ft_R, pt_R \leftarrow pt$
G02 AE $\leftarrow \{pt\}$
G03 SC $\leftarrow \emptyset$
G04 K$[\cdot] \leftarrow \cdot$
G05 $(ss, ps) \leftarrow \text{gen}(pt)$
G06 Invoke $\mathcal{A}$
G07 Lose

Oracle Evolveps
E10 evolveps$\langle ps \rangle$
E11 $pt_S \leftarrow pt_S + 1$

Oracle Enc
E00 $(k, c) \leftarrow \text{enc}(ps)$
E01 $cid := (pt_S, c)$
E02 Promise $cid \notin$ SC
E03 SC $\xleftarrow{\cup} \{cid\}$
E04 K$[cid] \leftarrow k$
E05 Return (k, c)

Oracle Expire
X00 expire$\langle ss \rangle$
X01 Promise $ft_R \in$ AE
X02 AE $\leftarrow$ AE $\setminus \{ft_R\}$
X03 $ft_R \leftarrow ft_R + 1$

Oracle Dec(pt, c)
D00 $k \leftarrow \text{dec}(ss, pt, c)$
D01 Promise $pt \in$ AE
D02 $cid := (pt, c)$
D03 If $cid \in$ SC:
D04 Promise $k =$ K$[cid]$
D05 $k \leftarrow \diamond$
D06 Return k

Oracle Evolvess
E20 evolvess$\langle ss \rangle$
E21 $pt_R \leftarrow pt_R + 1$
E22 AE $\xleftarrow{\cup} \{pt_R\}$

Fig. 2.15 Game FUNC for KeKEM.

a KeKEM is *functional* if the advantage $\mathbf{Adv}^{\text{FUNC}}(\mathcal{A}) := \max_{pt \in \mathbb{N}} \mathbb{P}[\text{FUNC}(pt, \mathcal{A})]$ is negligibly small for all realistic adversaries $\mathcal{A}$.

Security of KeKEM. We require of KeKEM schemes that they provide IND-CCA like confidentiality, with the strongest possible model for state corruption (incorporating forward secrecy). This is formalized via Games CONF^0, CONF^1 in Fig. 2.16, where oracles Challenge, Dec, Evolveps, Evolvess, Expire give access to the corresponding scheme algorithms (with oracle Challenge giving left-or-right access to algorithm enc), oracles ExposeS, ExposeR reveal user states, and oracle Decide serves the adversary for delivering a guess on challenge bit b.

We define $\mathbf{Adv}^{\text{CONF}}(\mathcal{A}) := \max_{pt \in \mathbb{N}} |\mathbb{P}[\text{CONF}^1(pt, \mathcal{A})] - \mathbb{P}[\text{CONF}^0(pt, \mathcal{A})]|$ as the advantage of an adversary $\mathcal{A}$ in the CONF game. We say the scheme provides confidentiality if the advantage $\mathbf{Adv}^{\text{CONF}}(\mathcal{A})$ that can be obtained by any realistic adversary $\mathcal{A}$ is negligible. Intuitively, the CONF games require that using a secret state for decapsulation only produces the correct key if the epoch had not yet been expired. Set XP indicates for which epochs the receiver has been exposed: Initially this set is empty [G05], but when the receiver is exposed, all active and future epochs ('AFE') are added to the set [E31,E33]. Set CH tracks for which epochs the adversary has issued a challenge. Initially this set is empty [G04], but an epoch is added to the set in a Challenge query [C05]. The adversary is not allowed to challenge an exposed epoch [C00] and similarly not allowed to expose an active or future challenged epoch [E32].

Construction. In Fig. 2.17 we provide a construction from a forward-secure KEM [36]. Note that the construction is straightforward with array SS$_R$ storing multiple secret

Game $\mathrm{CONF}^b(pt, \mathcal{A})$
G00 $pt_S \leftarrow pt$
G01 $ft_R, pt_R \leftarrow pt$
G02 $\mathrm{AE} \leftarrow \{pt\}$
G03 $\mathrm{SC} \leftarrow \emptyset$
G04 $\mathrm{CH} \leftarrow \emptyset$
G05 $\mathrm{XP} \leftarrow \emptyset$
G06 $(ss, ps) \leftarrow \mathrm{gen}(pt)$
G07 Invoke $\mathcal{A}$
G08 Lose

Oracle ExposeS
E10 Return ps

Oracle Decide(b')
D00 Stop with b'

Oracle Challenge
C00 Require $pt_S \notin \mathrm{XP}$
C01 $(k^0, c) \leftarrow \mathrm{enc}(ps)$
C02 $k^1 \leftarrow_\$ \mathcal{K}$
C03 $cid \coloneqq (pt_S, c)$
C04 $\mathrm{SC} \xleftarrow{\cup} \{cid\}$
C05 $\mathrm{CH} \xleftarrow{\cup} \{pt_S\}$
C06 Return (k^b, c)

Oracle Evolveps
E00 evolveps$\langle ps \rangle$
E01 $pt_S \leftarrow pt_S + 1$

Oracle Expire
X00 expire$\langle ss \rangle$
X01 $\mathrm{AE} \leftarrow \mathrm{AE} \setminus \{ft_R\}$
X02 $ft_R \leftarrow ft_R + 1$

Oracle $\mathrm{Dec}(pt, c)$
D10 $k \leftarrow \mathrm{dec}(ss, pt, c)$
D11 $cid \coloneqq (pt, c)$
D12 If $cid \in \mathrm{SC}$:
D13 $\quad k \leftarrow \diamond$
D14 Return k

Oracle Evolvess
E20 evolvess$\langle ss \rangle$
E21 $pt_R \leftarrow pt_R + 1$
E22 $\mathrm{AE} \xleftarrow{\cup} \{pt_R\}$

Oracle ExposeR
E30 $\mathrm{FE} \coloneqq [\![pt_R + 1 \,..\, \infty]\!]$
E31 $\mathrm{AFE} \coloneqq \mathrm{AE} \cup \mathrm{FE}$
E32 Require $\mathrm{CH} \cap \mathrm{AFE} = \emptyset$
E33 $\mathrm{XP} \xleftarrow{\cup} \mathrm{AFE}$
E34 Return ss

Fig. 2.16 Games $\mathrm{CONF}^0, \mathrm{CONF}^1$ for KeKEM.

states: one for each active epoch. Thus, while the secret state evolves to decapsulate for a new active, an old copy can be used to decapsulate for earlier epochs.

Proc gen(pt)
g00 $(ss', pk) \leftarrow \text{FSgen}$
g01 $\text{SS}_R[\cdot] \leftarrow \bot$
g02 $\text{SS}_R[pt] \leftarrow ss'$
g03 $ft_R, pt_R \leftarrow pt$
g04 $ss := (\text{SS}_R, ft_R, pt_R)$
g05 $pt_S \leftarrow pt$
g06 $ps := (pk, pt_S)$
g07 Return (ss, ps)

Proc enc(ps)
e00 $(k, c) \leftarrow \text{FSenc}(pk, pt_S)$
e01 Return (k, c)

Proc evolveps$\langle ps \rangle$
e10 $pt_S \leftarrow pt_S + 1$

Proc dec(ss, pt, c)
d00 Require $\text{SS}_R[pt] \neq \bot$
d01 $k \leftarrow \text{FSdec}(\text{SS}_R[pt], pt, c)$
d02 Return k

Proc evolvess$\langle ss \rangle$
e20 Require $\text{SS}_R[pt_R] \neq \bot$
e21 $ss' \leftarrow \text{SS}_R[pt_R]$
e22 $\text{FSupd}\langle ss' \rangle(pt_R)$
e23 $pt_R \leftarrow pt_R + 1$
e24 $\text{SS}_R[pt_R] \leftarrow ss'$

Proc expire$\langle ss \rangle$
x00 Require $\text{SS}_R[ft_R] \neq \bot$
x01 $\text{SS}_R[ft_R] \leftarrow \bot$
x02 $ft_R \leftarrow ft_R + 1$

Fig. 2.17 KeKEM construction. We use a forward-secure KEM (FSgen, FSenc, FSdec, FSupd) as a building block [36].

2.5 Interactive Primitives and KuBOOM

This section presents our Key-updatable Bidirectional Out-of-Order Messaging protocol, KuBOOM, in three steps. In Sec. 2.5.1 we first present a KuBOOM-Signature Scheme, which uses the USS introduced in Sec. 2.4.1 as building block. This scheme will be used by our final KuBOOM construction in a black box manner by calling its sign and vfy procedures on each message to add an authenticity layer. Next, we present a KuBOOM-KEM in Sec. 2.5.2. Our final KuBOOM construction will query the KuBOOM-KEM in a black box manner by calling its enc and dec procedures to obtain encryption keys for each message. The KuBOOM-KEM uses the KuKEM and KeKEM introduced in Sec. 2.4.2 and Sec. 2.4.3 as building blocks to ensure the KuBOOM scheme can achieve confidentiality with its keys. Hence, the KuBOOM construction will additionally invoke its upd procedure to reflect the passing of time and the expire procedure to indicate we no longer wish to be able to obtain 'old' decryption keys.

Despite the strong building blocks defined in Sec. 2.4, our KuBOOM protocols remain complex and involved. These difficulties stem from the data structures required to manage out-of-order delivery of ciphertexts. These data structures obscure the cryptographically novel core of our construction and render it difficult to interpret. Therefore, we have separated the authenticity tool and the confidentiality tool and present them in their own right. Note that this modularization implies certain data structures will be duplicated across each tool, but an implementation could consolidate them.

2.5.1 KuBOOM-Signature Scheme

In Sec. 2.5.3 we will use a *Key-updatable Bidirectional Out-of-Order Messaging Signature Scheme*, in short KuBOOM-Signature Scheme, to achieve authenticity for our KuBOOM construction. In this section we describe the inner workings of this cryptographic tool.

Syntax. A *KuBOOM-Signature Scheme* for a message space $\mathcal{M}$ consists of a state space $\mathcal{S}$, a signature space Σ, and algorithms init, sign, vfy and the timestamp decoder ts that recovers the logical time.

$$\text{init} \to \mathcal{S} \times \mathcal{S} \qquad \Sigma \to \text{ts} \to \mathbb{N} \qquad \mathcal{M} \to \text{sign}\langle\mathcal{S}\rangle \to \Sigma \qquad \mathcal{M} \times \Sigma \to \text{vfy}\langle\mathcal{S}\rangle \ .$$

Construction. We provide a construction for a KuBOOM-Signature Scheme in Fig. 2.18. The construction consists of four procedures: init, sign, vfy and ts. The init procedure initializes the states for two users `A` and `B`. The sign procedure is stateful and will output a signature σ for any message m, updating its state in the process. The

vfy procedure is also stateful and will verify any pair $(m, \sigma) \in \mathcal{M} \times \Sigma$. If σ is a correct signature on m, the state will update and vfy will return control to the caller. If the signature does not correctly verify, the vfy procedure will abort. The ts procedure returns the logical time.

On a very high level, sign generates fresh USS signing and verification states every iteration to recover from (potential) state exposures and signs the hash of its sent transcript, and vfy updates its signing state with the messages that have been received, so states will diverge if the adversary injects a message, while managing out of order delivery. We will now describe the variables and code lines in more detail.

Proc init
i00 For $u \in \{\mathtt{A}, \mathtt{B}\}$:
i01 $\quad lt_u \leftarrow 0$; $lt_u^* \leftarrow 0$
i02 $\quad \mathrm{S}_u[\cdot] \leftarrow \bot$; $\mathrm{V}_u[\cdot] \leftarrow \bot$
i03 $\quad (ss_u, vs_u^*) \leftarrow$ USS.gen
i04 $\quad \mathrm{P}_u \leftarrow \emptyset$
i05 $\quad \mathrm{AS}_u[\cdot] \leftarrow \bot$
i06 $\quad \mathrm{AS}_u[lt_u] \leftarrow H()$
i07 $\quad av_u \leftarrow H()$
i08 $\quad st_u \coloneqq (\ldots)$
i09 Return $(st_{\mathtt{A}}, st_{\mathtt{B}})$

Proc sign$\langle st_u \rangle(m)$
s00 $(ss, vs) \leftarrow$ USS.gen
s01 $h \leftarrow H(m \mathbin{\#} lt_u \mathbin{\#} \mathrm{P}_u \mathbin{\#} vs \mathbin{\#} \mathrm{S}_u[\![lt_u]\!])$
s02 $\sigma \leftarrow$ USS.sign$\langle ss_u \rangle(h)$
s03 $\mathrm{AS}_u[lt_u + 1] \leftarrow H(\mathrm{AS}_u[lt_u] \mathbin{\#} h \mathbin{\#} \sigma)$
s04 $\mathrm{S}_u[lt_u] \leftarrow (h, \mathrm{P}_u, vs, \sigma)$
s05 $\sigma \xleftarrow{\#} lt_u \mathbin{\#} \mathrm{P}_u \mathbin{\#} vs \mathbin{\#} \mathrm{S}_u[\![lt_u]\!]$
s06 $lt_u \leftarrow lt_u + 1$
s07 $(ss_u, \mathrm{P}_u) \leftarrow (ss, \emptyset)$
s08 Return σ

Proc ts(σ)
t00 Parse $\sigma \mathbin{\#} lt \mathbin{\#} \mathrm{P} \mathbin{\#} vs \mathbin{\#} \mathrm{S}[\![lt]\!] \leftarrow \sigma$
t01 Return lt

Proc vfy$\langle st_u \rangle(m, \sigma)$
v00 Parse $\sigma \mathbin{\#} lt \mathbin{\#} \mathrm{P} \mathbin{\#} vs \mathbin{\#} \mathrm{S}[\![lt]\!] \leftarrow \sigma$
v01 $h \leftarrow H(m \mathbin{\#} lt \mathbin{\#} \mathrm{P} \mathbin{\#} vs \mathbin{\#} \mathrm{S}[\![lt]\!])$
v02 While $lt_u^* \leq lt$:
v03 $\quad$ If $lt_u^* = lt$:
v04 $\quad\quad (\mathrm{P}', vs') \leftarrow (\mathrm{P}, vs)$
v05 $\quad\quad (h', \sigma') \leftarrow (h, \sigma)$
v06 $\quad$ Else:
v07 $\quad\quad (h', \mathrm{P}', vs', \sigma') \leftarrow \mathrm{S}[lt_u^*]$
v08 $\quad vs^* \leftarrow vs_{\bar{u}}^*$
v09 $\quad$ For $i \in \mathrm{P}'$:
v10 $\quad\quad$ USS.updvs$\langle vs^* \rangle(\mathrm{AS}_u[i])$
v11 $\quad$ Require USS.vfy(vs^*, h', σ')
v12 $\quad vs_{\bar{u}}^* \leftarrow vs'$
v13 $\quad av_u \leftarrow H(av_u \mathbin{\#} h' \mathbin{\#} \sigma')$
v14 $\quad \mathrm{V}_u[lt_u^*] \leftarrow (h', \sigma')$
v15 $\quad$ USS.updss$\langle ss_u \rangle(av_u)$
v16 $\quad lt_u^* \leftarrow lt_u^* + 1$
v17 $\quad \mathrm{P}_u \xleftarrow{\cup} \{lt_u^*\}$
v18 If $lt_u^* > lt$:
v19 $\quad$ Require $\mathrm{V}_u[lt] \neq \diamond$
v20 $\quad$ Require $\mathrm{V}_u[lt] = (h, \sigma)$
v21 $\mathrm{V}_u[lt] \leftarrow \diamond$

Fig. 2.18 KuBOOM-Signature Scheme construction. We use an updatable signature scheme (USS) as building block. Function H is assumed to be a collision-resistant hash function. The vfy procedure aborts if parsing fails.

For each user $u \in \{\mathtt{A}, \mathtt{B}\}$ we initialize the signing index lt_u and the verifying index lt_u^* [i01], and the arrays S_u and V_u, which will store information about signed and verified messages, respectively [i02]. We generate pairs of USS signing and verification states [i03]

and initialize the set P_u of messages processed by the current signing state to be empty [i04]. We initialize the accumulated signed transcript AS_u [i05], set the first index [i06] and initialize the accumulated verified transcript av_u [i07]. Finally, we store everything in the users' states [i08].

The sign procedure first generates new USS signing and verification states [s00]. Next, it computes the hash of the message m, the signing index lt_u, the set of processed messages P_u, the verification state vs and the array S_u [s01]. It signs the hash using its old signing state [s02]. It accumulates the new hash and signature in AS_u [s03] and stores the hash, processed set, verification state and signature in S_u [s04]. The signing index, processed set, verification state and array S_u are appended to the signature [s05]. It increments the signing index lt_u [s06] and stores the new signing state along with an empty processed set [s07], before returning the signature [s08].

The vfy procedure parses the additional information embedded in the signature [v00] and recomputes the hash [v01]. If the verifying index lt_u^* is less or equal than lt, the verifier will iteratively check signatures until it catches up [v02–v17]. To be concrete, if $lt_u^* = lt$ it will use the current value for the hash and signature [v05] or if $lt_u^* < lt$ it will obtain these values from $\mathrm{S}[lt_u^*]$ [v07]. It will update a copy of its verification state for all indices the signer has processed since generating its signing state [v08–v10] and verify the signature [v11]. Note it uses the transcript for its signed messages to update its verification state, which should match the transcript for the verified messages the signer has used to update its signing state. If signature verification passes, it will replace the verification state [v12]. Note that if USS.vfy failed the verification state remains unchanged, as if [v09–v10] were never executed. Next, it accumulates the hash and signature in its verified transcript av_u [v13], stores the hash and signature in $\mathrm{V}_u[lt_u^*]$ for later comparison [v14] and it will update its signing state ss_u with av_u [v15]. It increments the index lt_u^* [v16] and add lt_u^* to P_u to indicate it has processed this message into its signing state [v17]. If the verifying index lt_u^* was strictly greater than lt, the verifier will check if an entry exists for this index [v19] and compare whether it is equal to the value of the hash and signature [v20]. At last, the verifier will remove the entry in V_u for index lt to prevent double delivery [v21].

Note for simplicity we omit code lines to 'clean up' variables that are no longer needed. These lines are not required for security, but would help for efficiency. For example, if a party learns its peer has processed signature i, it will no longer have to include the first i entries of S_u in its next signature.

2.5.2 KuBOOM-KEM

In Sec. 2.5.3 we will use a *Key-updatable Bidirectional Out-of-Order Messaging KEM*, in short KuBOOM-KEM, to help achieve confidentiality for our KuBOOM construction. In this section we describe the inner workings of this cryptographic tool.

Syntax. A *KuBOOM-KEM* for a key space $\mathcal{K}$ consists of a state space $\mathcal{S}$, a ciphertext space $\mathcal{C}$, and algorithms init, upd, expire, enc, dec and the timestamp decoder ts that recovers the logical and physical time.

$$\text{init} \to \mathcal{S} \times \mathcal{S} \qquad \text{upd}\langle\mathcal{S}\rangle \qquad \text{expire}\langle\mathcal{S}\rangle \qquad \mathcal{C} \to \text{ts} \to \mathbb{N} \times \mathbb{N}$$

$$\mathcal{AD} \to \text{enc}\langle\mathcal{S}\rangle \to \mathcal{K} \times \mathcal{C} \qquad \mathcal{AD} \times \mathcal{C} \to \text{dec}\langle\mathcal{S}\rangle \to \mathcal{K} \ .$$

Construction. Internally our KuBOOM-KEM construction will invoke the KuKEM primitive introduced in Sec. 2.4.2, the KeKEM primitive introduced in Sec. 2.4.3, and a secure KEM combiner K such that if at least one of the input keys is indistinguishable from a uniformly random string of equal length, then so is the output key. In this thesis we will consider K a random oracle. An implementation could use the CCA secure combiner presented in [53].

We noted both our KuKEM and KeKEM building block can be built generically from hierarchical identity-based encryption (HIBE, [52]). This strong component, while inefficient, should come as no surprise as it has already been proposed by [56] and [86] in the much simpler setting where every message is always delivered, and always in order. Moreover, recent work [12] shows that if an exposure additionally reveals the random coins used for the next send operation, the use of KuKEM is required to achieve confidentiality. They hypothesize the same implication holds without revealing the random coins and provide a strong intuition, but a formal proof remains an open problem.

We remark that both our KuKEM and KeKEM can be built from a single HIBE instance if one immediately delegates the master secret key to a 'KuKEM identity' and to a 'KeKEM identity'. We avoid doing so for two reasons. First of all, these primitives correspond to two perpendicular security goals. It is conceptually easier to grasp if we do not intertwine them. Secondly, KeKEM can be built from a forward-secure KEM, which is a simpler primitive than the HIBE-KEM used for KuKEM. Thus it may also be more efficient to separate them.

We provide a construction for a KuBOOM-KEM in Fig. 2.19. The construction consists of six procedures: init, enc, dec, expire, upd and ts. A correct decryption procedure dec is determined by the encryption procedure: it mirrors the operations in

enc. As deriving the dec procedure is a rather vacuous technical exercise we defer it to focus on the more interesting cryptographic procedures first. For completeness we describe the dec procedure at the end of this section. Note the ts procedure simply parses the timestamps embedded in each ciphertext and returns the logical and physical (creation) time of a ciphertext.

The construction is quite technical but the general idea is to generate a new KuKEM and a new KeKEM instance with every enc invocation for post-compromise security. We update the KeKEM for forward secrecy in physical time, and the KuKEM for forward secrecy in logical time. The enc procedure will output a key dependent on the output of the KuKEM encapsulation procedure, the KeKEM encapsulation procedure and the associated data input.

We remark the physical time updates must be a separate primitive as simply updating the KuKEM would render the users out-of-sync. For example, consider the scenario where Alice sends a message, updating her KuKEM. Now physical time advances and both Alice and Bob would update their KuKEM. Finally, Bob receives Alice's message and updates his KuKEM. Clearly the updates have occurred in a different order, hence functionality will fail.

We note our security notion implies ciphertexts must contain information about prior ciphertexts. To see that ciphertexts cannot be independent, consider an adversary that exposes Alice and creates two ciphertexts. The adversary will deliver the second ciphertext to Bob, rendering Bob out-of-sync. Now the adversary can challenge Alice, making her send her first ciphertext, and since Bob is out-of-sync, expose Bob. If Bob were able to decrypt any ciphertext with logical index 1, the adversary could now decrypt Alice's challenge ciphertext and win the confidentiality game. Hence, the second ciphertext must 'pin' the first.

We achieve this with the KEM/DEM encryption paradigm. The enc procedure will embed past KuKEM ciphertexts in the current ciphertext. When receiving a ciphertext, the dec procedure will decapsulate all embedded KuKEM ciphertexts, store the DEM keys and destroy its capability to decapsulate again. Reconsidering our example above, Bob is now only able to decrypt the first ciphertext if it was encrypted with the same DEM key he obtained from the second ciphertext, and Bob has no capability to decapsulate another KuKEM ciphertext. The probability that Alice and the adversary had generated the same KuKEM ciphertext for the first ciphertext is negligible.

We now discuss the procedures in more detail, starting with init. For each user the init procedure initializes a sending index lt_u, a receiving index lt_u^*, the first physical time that is still recoverable ft_u and the current physical time pt_u [i01–i02]. It initializes

the array AS for the accumulated sent transcript and AR for the accumulated received transcript [i03–i06]. The accumulated transcripts will be used to update the KuKEM states, ensuring the user states diverge when users go out-of-sync. Because ciphertexts may be delivered out-of-order, or not at all, each user will be maintaining several instances of each primitive, ready to decapsulate ciphertexts for any of them. However, it will always encapsulate to the latest one. Hence we initialize storage for multiple secret states, but only one public state, and we store the first KeKEM and KuKEM instance [i07–i10]. Finally, we initialize the array KC to store KuKEM ciphertexts [i11] and the array DK to store DEM keys [i12], as described in the general construction overview.

The enc procedure encapsulates keys for both the KeKEM and the KuKEM [e00–e01], and stores the KuKEM ciphertext in KC, along with its receiver index lt_u^*, indicating which public states were used for encapsulation [e02]. Next, it generates a new instance for both the KeKEM and the KuKEM [e03–e04]. It will immediately update the secret state for the KuKEM with the received transcript [e05], as the adversary is allowed unrestricted expose queries if we are out-of-sync. The enc procedure combines the KEM ciphertexts into one ciphertext, adds the freshly generated public states, and includes the indices and the sending transcript such that the receiver can correctly update its state [e06–e07]. Subsequently, it uses the KEM-combiner K to produce a key, using the associated data and ciphertext as context [e09]. Finally, it increments the sending index lt_u [e10], accumulates the associated data and ciphertext into its transcript [e11] and updates its public KuKEM state with it [e12].

The upd procedure is quite straightforward: it simply updates the public state and evolves the secret states for all its KeKEM instances as physical time advances. Similarly, the expire procedure will update all secret states.

Note that for simplicity we have omitted code lines to 'clean up' variables that are no longer needed. These lines are not required for security, but would help for efficiency. For example when a user has either received or expired all messages encapsulated for its i-th KeKEM and KuKEM instance (it knows an upper bound for messages encapsulated to its i-th instance when receiving a message encapsulated to instance $j > i$) , it can drop instance i, as later keys will always be encapsulated to later instances. As another example we remark that, after receiving an acknowledgement from the other user they have received message i, a user would no longer have to embed all their KuKEM ciphertexts for indices less than or equal to i in their current ciphertext.

Here we provide a detailed description of the dec procedure. As indicated, it is mainly a technical exercise to handle the out-of-order delivery correctly.

The dec procedure stores a copy of the associated data and the ciphertext [d00] to be used as context for the combiner later in the procedure [d25] and to accumulate into its received transcript [d22]. It will parse all the information the sender has embedded in the ciphertext [d01–d02], and obtain the sender's latest received index to identify to which KeKEM and KuKEM instances the keys were encapsulated [d03]. Next, it requires the sending user is responding to a sending index the receiving user has actually reached and a physical time that has not yet expired [d04–d05]. If the sending index lt is greater than what has been received so far [d06], this means the embedded public states are the latest. Hence we will overwrite our public states [d07,d10] and update them from creation time to the current time [d09,d12]. The receiver will iteratively catch up from the last received index to the current index [d13–d22]. First, it will decapsulate the KuKEM ciphertext and store the DEM key [d14,d15]. Next, the receiver index is updated [d16]. For each KuKEM, it will update the secret state [d22], obtaining the required update information either from AS [d18] or compute the accumulated received transcript itself for the current index [d20]. Now the receiver is up to date, we decapsulate the KeKEM ciphertext to obtain k_0 and retrieve k_1 from DK [d23–d24]. The KEM-combiner K is used to combine k_0, k_1 and adc into one key [d25]. Finally, we need to clear the DEM key from memory, such that an adversary cannot decrypt old ciphertexts after an exposure [d26].

Proc init
i00 For $u \in \{\mathtt{A}, \mathtt{B}\}$:
i01 $\quad lt_u \leftarrow 0;\ lt_u^* \leftarrow 0$
i02 $\quad ft_u \leftarrow 0;\ pt_u \leftarrow 0$
i03 $\quad \mathrm{AS}_u[\cdot] \leftarrow \bot$
i04 $\quad \mathrm{AR}_u[\cdot] \leftarrow \bot$
i05 $\quad \mathrm{AS}_u[lt_u] \leftarrow H()$
i06 $\quad \mathrm{AR}_u[lt_u^*] \leftarrow H()$
i07 $\quad E_u[\cdot] \leftarrow \bot$
i08 $\quad U_u[\cdot] \leftarrow \bot$
i09 $\quad (E_u[lt_u], \varepsilon_{\bar{u}}^*) \leftarrow \mathrm{ke.gen}(pt_u)$
i10 $\quad (U_u[lt_u], \upsilon_{\bar{u}}^*) \leftarrow \mathrm{ku.gen}$
i11 $\quad \mathrm{KC}_u[\cdot] \leftarrow \bot$
i12 $\quad \mathrm{DK}_u[\cdot] \leftarrow \bot$
i13 $\quad st_u \coloneqq (\ldots)$
i14 Return $(st_{\mathtt{A}}, st_{\mathtt{B}})$

Proc $\mathrm{enc}\langle st_u \rangle(ad)$
e00 $(k_0, c_0) \leftarrow \mathrm{ke.enc}(\varepsilon_u^*)$
e01 $(k_1, c_1) \leftarrow \mathrm{ku.enc}(\upsilon_u^*)$
e02 $\mathrm{KC}_u[lt_u] \leftarrow (lt_u^*, c_1)$
e03 $(E_u[lt_u], \varepsilon) \leftarrow \mathrm{ke.gen}(pt_u)$
e04 $(U_u[lt_u], \upsilon) \leftarrow \mathrm{ku.gen}$
e05 $\mathrm{ku.updss}\langle U_u[lt_u] \rangle(\mathrm{AR}_u[lt_u^*])$
e06 $c \leftarrow lt_u \mathbin{\#} pt_u \mathbin{\#} \varepsilon \mathbin{\#} \upsilon$
e07 $c \xleftarrow{+} c_0 \mathbin{\#} \mathrm{KC}_u[*] \mathbin{\#} \mathrm{AS}_u[*]$
e08 $adc \leftarrow ad \mathbin{\#} c$
e09 $k \leftarrow K(k_0, k_1; adc)$
e10 $lt_u \leftarrow lt_u + 1$
e11 $\mathrm{AS}_u[lt_u] \leftarrow H(\mathrm{AS}_u[lt_u - 1] \mathbin{\#} adc)$
e12 $\mathrm{ku.updps}\langle \upsilon_u^* \rangle(\mathrm{AS}[lt_u])$
e13 Return (k, c)

Proc $\mathrm{upd}\langle st_u \rangle$
u00 $pt_u \leftarrow pt_u + 1$
u01 $\mathrm{ke.evolveps}\langle \epsilon_u^* \rangle$
u02 For $i \in [lt]_u$:
u03 $\quad \mathrm{ke.evolvess}\langle E_u[i] \rangle$

Proc $\mathrm{ts}(c)$
t00 Parse $lt \mathbin{\#} pt \mathbin{\#} \ldots \leftarrow c$
t01 Return (lt, pt)

Proc $\mathrm{dec}\langle st_u \rangle(ad, c)$
d00 $adc \leftarrow ad \mathbin{\#} c$
d01 Parse $lt \mathbin{\#} pt \mathbin{\#} \varepsilon \mathbin{\#} \upsilon \mathbin{\#} c \leftarrow c$
d02 Parse $c_0 \mathbin{\#} \mathrm{KC}[*] \mathbin{\#} \mathrm{AS}[*] \leftarrow c$
d03 $(lt^*, \ldots) \leftarrow \mathrm{KC}[lt]$
d04 Require $lt^* \le lt_u$
d05 Require $ft_u \le pt$
d06 If $lt_u^* < lt$:
d07 $\quad \varepsilon_u^* \leftarrow \varepsilon'$
d08 $\quad$ For $t \in [\![pt \,..\, pt_u]\!]$:
d09 $\quad\quad \mathrm{ke.evolveps}\langle \varepsilon_u^* \rangle$
d10 $\quad \upsilon_u^* \leftarrow \upsilon'$
d11 $\quad$ For $i \leftarrow lt^*$ to lt_u:
d12 $\quad\quad \mathrm{ku.updps}\langle \upsilon' \rangle(\mathrm{AS}_u[i])$
d13 While $lt_u^* \le lt$:
d14 $\quad (lt^*, c^*) \leftarrow \mathrm{KC}[lt_u^*]$
d15 $\quad \mathrm{DK}[lt_u^*] = \mathrm{ku.dec}(U_u[lt^*], c^*)$
d16 $\quad lt_u^* \leftarrow lt_u^* + 1$
d17 $\quad$ If $lt_u^* \le lt$:
d18 $\quad\quad \mathrm{AR}_u[lt_u^*] \leftarrow \mathrm{AS}[lt_u^*]$
d19 $\quad$ Else:
d20 $\quad\quad \mathrm{AR}_u[lt_u^*] \leftarrow H(\mathrm{AR}_u[lt_u^* - 1] \mathbin{\#} adc)$
d21 $\quad$ For $i \in [lt_u]$:
d22 $\quad\quad \mathrm{ku.updss}\langle U_u[i] \rangle(\mathrm{AR}_u[lt_u^*])$
d23 $k_0 \leftarrow \mathrm{ke.dec}(E_u[lt^*], pt, c_0')$
d24 $k_1 \leftarrow \mathrm{DK}[lt]$
d25 $k \leftarrow K(k_0, k_1; adc)$
d26 $\mathrm{DK}[lt] \leftarrow \bot$
d27 Return k

Proc $\mathrm{expire}\langle st_u \rangle$
x00 $ft_u \leftarrow ft_u + 1$
x01 For $i \in [lt_u]$:
x02 $\quad \mathrm{ke.expire}\langle E_u[i] \rangle$

Fig. 2.19 KuBOOM-KEM construction. Building blocks are a KeKEM, whose algorithms are prefixed with 'ke.', a KuKEM, whose algorithms are prefixed with 'ku.' and a KEM combiner K.

2.5.3 KuBOOM Construction

We first introduce a functional protocol and discuss it in detail before delving into the full KuBOOM construction that achieves authenticity and confidentiality. The functional protocol consists of all the unmarked code lines in Fig. 2.20. The protocol has four procedures: the initialization procedure init, which initializes the users' initial states; the sending procedure send, which takes a state, associated data and a message, updates the state and outputs a ciphertext; the receiving procedure recv, which takes a state, associated data and a ciphertext, updates the state and outputs a message; and the time progression algorithm tick, which updates the state.

For each user u, the init procedure initializes the logical time lt_u and lt_u^* [i03], the physical time pt_u [i04], the set of received indices RI_u [i04], the set of received timestamps RT_u [i05], the set of received acknowledgements RA_u [i05], and the arrays of hashed sent ciphertexts HS_u and hashed received ciphertexts HR_u [i06]. The tick procedure increments the user's physical time pt_u [u00].

The send procedure takes associated data ad and message m as input. It creates context ctx which includes the user's current time (lt_u, pt_u), the hashes of previously sent ciphertexts $\mathrm{HS}_u[*]$ and the set of received indices RI_u [s00]. The context ctx together with the message m will form the ciphertext c [s07]. Finally, it stores the hash $H(ad \mathbin{+\!\!+} c)$ of the associated data and the ciphertext [s10], increments the logical time lt_u [s11] and returns the ciphertext [s12].

The recv procedure first hashes the ciphertext [r00] and subsequently parses it to obtain the message m [r02] and the context variables lt, pt, $\mathrm{HS}[*]$ and R [r05]. Now, recall that 'Require C' is short for 'If $\neg C$: Abort'. Thus the recv procedure performs four sanity checks to guarantee functionality. (1) A ciphertext has not yet been received for this logical time lt [r08]. (2) The ciphertext is fresh, that is the Δ difference between its physical creation time pt and the user's time pt_u is 'small' [r09]. (3) Time is monotonic: a message that is newer in logical time must be newer in physical time [r10]. (4) Only sent messages can be acknowledged [r11]: Bob cannot acknowledge having received a message that Alice never sent. Next, recv handles the out-of-order delivery. While u's receiving index lt_u^* is smaller or equal than lt, it will iteratively update its array HR_u with the received hashes that it obtains from $\mathrm{HS}[*]$ or the current ciphertext itself [r12–r15]. If u's receiving index is greater than lt, it will require the hash h of the current ciphertext is equal to the stored value for that index $\mathrm{HR}_u[lt]$ [r16]. Finally, it will update its set of received indices RI_u [r17], its set of received timestamps RT_u [r18], its set of received acknowledgements RA_u [r19], and return (RA_u, m) [r22].

Proc init
- ○ i00 $(st_{\mathtt{A}}^{\mathrm{BS}}, st_{\mathtt{B}}^{\mathrm{BS}}) \leftarrow \mathrm{BS.init}$
- ● i01 $(st_{\mathtt{A}}^{\mathrm{BK}}, st_{\mathtt{B}}^{\mathrm{BK}}) \leftarrow \mathrm{BK.init}$
- i02 For $u \in \{\mathtt{A}, \mathtt{B}\}$:
- i03 $\quad lt_u \leftarrow 0;\ lt_u^* \leftarrow 0$
- i04 $\quad pt_u \leftarrow 0;\ \mathrm{RI}_u \leftarrow \emptyset$
- i05 $\quad \mathrm{RT}_u \leftarrow \emptyset;\ \mathrm{RA}_u \leftarrow \emptyset$
- i06 $\quad \mathrm{HS}_u[\cdot] \leftarrow \bot;\ \mathrm{HR}_u[\cdot] \leftarrow \bot$
- i07 $\quad st_u \coloneqq (\ldots)$
- i08 Return $(st_{\mathtt{A}}, st_{\mathtt{B}})$

Proc send$\langle st_u \rangle(ad, m)$
- s00 $\mathrm{ctx} \leftarrow lt_u \mathbin{\#} pt_u \mathbin{\#} \mathrm{HS}_u[*] \mathbin{\#} \mathrm{RI}_u$
- ○ s01 $(sk, vk) \leftarrow \mathrm{OTS.gen}$
- ○ s02 $\sigma_1 \leftarrow \mathrm{BS.sign}\langle st_u^{\mathrm{BS}} \rangle(vk)$
- ○ s03 $\mathrm{ctx} \xleftarrow{\#} vk \mathbin{\#} \sigma_1$
- ● s04 $(k, c') \leftarrow \mathrm{BK.enc}\langle st_u^{\mathrm{BK}} \rangle(vk)$
- ● s05 $\mathrm{ctx} \xleftarrow{\#} c'$
- ● s06 $m \leftarrow \mathrm{E.enc}(k, m)$
- s07 $c \leftarrow \mathrm{ctx} \mathbin{\#} m$
- ○ s08 $\sigma_2 \leftarrow \mathrm{OTS.sign}(sk, ad \mathbin{\#} c)$
- ○ s09 $c \xleftarrow{\#} \sigma_2$
- s10 $\mathrm{HS}_u[lt_u] \leftarrow H(ad \mathbin{\#} c)$
- s11 $lt_u \leftarrow lt_u + 1$
- s12 Return c

Proc ts(c)
- t00 Parse $lt_u \mathbin{\#} pt_u \mathbin{\#} \ldots \leftarrow c$
- t01 Return (lt_u, pt_u)

Proc tick$\langle st_u \rangle$
- u00 $pt_u \leftarrow pt_u + 1$
- ● u01 $\mathrm{BK.upd}\langle st_u^{\mathrm{BK}} \rangle$
- ● u02 If $\Delta(0, pt_u) > \delta$: $\mathrm{BK.expire}\langle st_u^{\mathrm{BK}} \rangle$

Proc recv$\langle st_u \rangle(ad, c)$
- r00 $h \leftarrow H(ad \mathbin{\#} c)$
- ○ r01 $c \mathbin{\#} \sigma_2 \leftarrow c$
- r02 Parse $\mathrm{ctx} \mathbin{\#} m \leftarrow c$
- ● r03 Parse $\mathrm{ctx} \mathbin{\#} c' \leftarrow \mathrm{ctx}$
- ○ r04 Parse $\mathrm{ctx} \mathbin{\#} vk \mathbin{\#} \sigma_1 \leftarrow \mathrm{ctx}$
- r05 Parse $lt \mathbin{\#} pt \mathbin{\#} \mathrm{HS}[*] \mathbin{\#} R \leftarrow \mathrm{ctx}$
- ○ r06 $\mathrm{BS.vfy}\langle st_u^{\mathrm{BS}} \rangle(vk, \sigma_1)$
- ○ r07 $\mathrm{OTS.vfy}(vk, ad \mathbin{\#} c, \sigma_2)$
- r08 Require $lt \notin \mathrm{RI}_u$
- r09 Require $\Delta(pt, pt_u) \leq \delta$
- r10 Require $\mathrm{RT}_u \cup \{(lt, pt)\}$ monotone
- r11 Require $R \subseteq [\![lt_u]\!]$
- r12 While $lt_u^* \leq lt$:
- r13 $\quad$ If $lt_u^* < lt$: $\mathrm{HR}_u[lt_u^*] \leftarrow \mathrm{HS}[lt_u^*]$
- r14 $\quad$ Else: $\mathrm{HR}_u[lt_u^*] \leftarrow h$
- r15 $\quad lt_u^* \leftarrow lt_u^* + 1$
- r16 If $lt_u^* > lt$: Require $\mathrm{HR}_u[lt] = h$
- r17 $\mathrm{RI}_u \xleftarrow{\cup} \{lt\}$
- r18 $\mathrm{RT}_u \xleftarrow{\cup} \{(lt, pt)\}$
- r19 $\mathrm{RA}_u \xleftarrow{\cup} R$
- ● r20 $k \leftarrow \mathrm{BK.dec}\langle st_u^{\mathrm{BK}} \rangle(vk, c')$
- ● r21 $m \leftarrow \mathrm{E.dec}(k, m)$
- r22 Return (RA_u, m)

Fig. 2.20 The functional construction consists of the unmarked lines. The authentic construction adds the lines marked with ○. The KuBOOM construction consists of all lines. BS is the KuBOOM-Signature Scheme construction in Fig. 2.18, BK is the KuBOOM-KEM construction in Fig. 2.19, OTS is a (one-time) signature scheme as defined in Sec. 1.5.2, and E is a symmetric encryption scheme as defined in Sec. 1.5.1.

We extend the functional protocol to an authentication protocol by including the lines marked with ○. The init procedure now initializes a KuBOOM-Signature Scheme BS [i00]. The send procedure generates a fresh one-time signature key pair (sk, vk) [s01], and calls BS.sign to obtain a signature σ_1 on the verification key vk [s02]. We add vk and σ_1 to the context ctx [s03]. We use the signing key sk to sign the associated data ad and ciphertext c [s08], and append the signature σ_2 to c [s09]. The recv procedure will parse the newly added signatures and verification key [r01,r04]. It will first verify the

signature on the verification key vk by calling BS.vfy [r06]. Then it uses vk to verify the signature on the associated data and ciphertext [r07].

It may appear peculiar not to sign the ciphertext directly with the KuBOOM-Signature Scheme signature. However, this design decision is made to simplify the confidentiality construction. If we sign the ciphertext directly, the adversary could expose the user to obtain its signing key and generate a new signature for the ciphertext. Indeed, this would not break authenticity as the forgery is trivial. Nonetheless, if the adversary submits the ciphertext to the Recv oracle with a different signature, the oracle will decrypt and return the (challenge) message. Now, because the one-time signature key pair is generated during the send procedure, it cannot be exposed. Thus, if the adversary succeeds in creating a valid but different signature, this would break the strong unforgeability property.

This brings us to the lines marked with •. Including these lines provides confidentiality, resulting in our KuBOOM protocol. The init procedure now also initializes a KuBOOM-KEM BK [i01]. The send procedure provides the KuBOOM-KEM with the verification key as context when requesting (k, c') [s04], appends c' to the context [s05], and uses k to encrypt the message [s06].

The adversary could have exposed the sender's state and created a (trivial) forgery by generating its own one-time signature pair. The Recv oracle would accept the ciphertext and attempt to decrypt it. Therefore, it is critical for confidentiality that the key derivation is dependent on the verification key [s04]. The recv procedure parses the newly added c' [r03] and inputs it, along with vk, to BK.dec to retrieve k [r20]. Subsequently, recv uses k to decrypt m [r21].

The tick procedure now calls BK.upd [u01] because its state must advance over time, even when no messages are exchanged, to achieve forward secrecy in physical time. Once time has advanced δ times it will start calling BK.expire [u02] to indicate we no longer desire to be able to decrypt 'old' messages. Neither of these procedures require the physical time as input because they advance linearly over time, with the expire procedure lagging behind the update procedure. This completes the description of our KuBOOM protocol in Fig. 2.20.

Our construction provides authenticity (Thm 2.6.1) and confidentiality (Thm 2.6.4). We formalize the theorems and provide the security proofs in Section 2.6.

2.6 Security Proofs

In this section we formalize theorems and prove the security of our constructions. We want to utilize our modular approach and prove the authenticity of our KuBOOM construction

via the authenticity of the KuBOOM-Signature Scheme. However, we have not yet formally defined an authenticity game for the latter. This game is almost equivalent to the authenticity game AUTH presented in Sec. 2.2 but adapted for message/signature pairs instead of associated data/ciphertext pairs and without a Tick oracle. We provide the authenticity game in Fig. 2.21 for completeness.

Game AUTH($\mathcal{A}$)
G00 For $u \in \{\mathtt{A}, \mathtt{B}\}$:
G01 $\quad lt_u \leftarrow 0$
G02 $\quad is_u \leftarrow \mathtt{T}$
G03 $\quad \mathrm{SC}_u \leftarrow \emptyset$
G04 $\quad \mathrm{CERT}_u \leftarrow \emptyset$
G05 $\quad \mathrm{VF}_u[\cdot] \leftarrow [\![\infty]\!]$
G06 $\quad \mathrm{AU}_u \leftarrow [\![\infty]\!]$
G07 $(st_{\mathtt{A}}, st_{\mathtt{B}}) \leftarrow \mathrm{init}$
G08 Invoke $\mathcal{A}$
G09 Lose

Oracle Sign(u, m)
S00 $\sigma \leftarrow \mathrm{sign}\langle st_u \rangle(m)$
S01 If is_u:
S02 $\quad \mathrm{SC}_u \xleftarrow{\cup} \{(m, \sigma)\}$
S03 $lt_u \leftarrow lt_u + 1$
S04 Return σ

Oracle Expose(u)
E00 If is_u:
E01 $\quad \mathrm{VF}_u[\![lt_u]\!] \xleftarrow{\cap} [\![lt_u]\!]$
E02 $\quad \mathrm{AU}_u \xleftarrow{\cap} [\![lt_u]\!]$
E03 Return st_u

Oracle Vfy(u, m, σ)
V00 $lt \leftarrow \mathrm{ts}(\sigma)$
V01 $\mathrm{vfy}\langle st_u \rangle(m, \sigma)$
V02 If $(m, \sigma) \in \mathrm{SC}_{\bar{u}}$:
V03 $\quad$ If is_u:
V04 $\quad\quad \mathrm{CERT}_u \xleftarrow{\cup} [lt]$
V05 $\quad\quad \mathrm{AU}_{\bar{u}} \xleftarrow{\cup} \mathrm{VF}_{\bar{u}}[lt]$
V06 If $(m, \sigma) \notin \mathrm{SC}_{\bar{u}}$:
V07 $\quad$ If is_u:
V08 $\quad\quad$ Reward $lt \in \mathrm{AU}_{\bar{u}}$
V09 $\quad$ Reward $lt \in \mathrm{CERT}_u$
V10 $\quad is_u \leftarrow \mathtt{F}$

Fig. 2.21 KuBOOM AUTH game.

Similarly, we need to specify a confidentiality notion for the KuBOOM-KEM in order to make rigorous statements about it. We provide the confidentiality game in Fig. 2.22. This game is almost equivalent to the confidentiality game presented in Sec. 2.2 but adapted for indistinguishability of keys instead of messages. Moreover, the Tick oracle has been split in an Upd and Expire oracle, granting the adversary more control. Most notably however, is the added line 'Require is_u' in the Challenge oracle. Since *poisoned* implies out-of-sync, the line 'Penalize $poisoned_u$' has become obsolete in the Challenge oracle in Fig. 2.22. Hence, this line, and all the lines and variables used for tracking the *poisoned* flag, have been removed from the game.

2.6.1 KuBOOM achieves Authenticity

We prove our KuBOOM construction provides authenticity by reducing the problem to the security of a regular signature scheme (Sec. 1.5.2) and the security of a USS (Sec. 2.4.1). In order to do so, in Lemma 2.6.2 we first reduce the security of our KuBOOM construction to the security of regular signatures and the security of our KuBOOM-Signature Scheme introduced in Sec. 2.5.1. Subsequently, we reduce the security of KuBOOM-Signature Scheme to the security of USS in Lemma 2.6.3.

Game $\text{CONF}^b(\mathcal{A})$

G00 For $u \in \{\mathtt{A},\mathtt{B}\}$:
G01 $\quad lt_u \leftarrow 0$
G02 $\quad pt_u \leftarrow 0$
G03 $\quad is_u \leftarrow \mathtt{T}$
G04 $\quad \text{SC}_u \leftarrow \emptyset$
G05 $\quad lx_u \leftarrow 0$
G06 $\quad \text{CH}_u \leftarrow \emptyset$
G07 $\quad xp_u \leftarrow \mathtt{F}$
G08 $(st_{\mathtt{A}}, st_{\mathtt{B}}) \leftarrow \text{init}$
G09 Invoke $\mathcal{A}$
G10 Lose

Oracle $\text{Expose}(u)$

E00 Require $\text{CH}_{\bar{u}} = \emptyset$
E01 If is_u:
E02 $\quad lx_u \leftarrow lt_u$
E03 $\quad xp_u \leftarrow \mathtt{T}$
E04 Return st_u

Oracle $\text{Upd}(u)$

U00 $\text{upd}\langle st_u \rangle$
U01 $pt_u \leftarrow pt_u + 1$

Oracle $\text{Expire}(u)$

T00 $\text{expire}\langle st_u \rangle$
T01 $\text{CH}_{\bar{u}} \xleftarrow{\cap} \text{ITC}(u)$

Oracle $\text{Challenge}(u, ad)$

C00 Require is_u
C01 Penalize $xp_{\bar{u}}$
C02 $(k^0, c) \leftarrow \text{enc}\langle st_u \rangle(ad)$
C03 $k^1 \leftarrow_{\$} \mathcal{K}$
C04 $\text{SC}_u \xleftarrow{\cup} \{(ad, c)\}$
C05 If $is_{\bar{u}}$: $\text{CH}_u \xleftarrow{\cup} \{lt_u\}$
C06 $lt_u \leftarrow lt_u + 1$
C07 Return (k^b, c)

Oracle $\text{Decide}(b')$

D00 Stop with b'

Oracle $\text{Enc}(u, ad)$

S00 $(k, c) \leftarrow \text{enc}\langle st_u \rangle(ad)$
S01 If is_u:
S02 $\quad \text{SC}_u \xleftarrow{\cup} \{(ad, c)\}$
S03 $lt_u \leftarrow lt_u + 1$
S04 Return (k, c)

Oracle $\text{Dec}(u, ad, c)$

R00 $(lt, pt) \leftarrow \text{ts}(c)$
R01 $k \leftarrow \text{dec}\langle st_u \rangle(ad, c)$
R02 If $(ad, c) \in \text{SC}_{\bar{u}}$:
R03 $\quad$ If is_u:
R04 $\quad\quad$ If $lt \geq lx_{\bar{u}}$:
R05 $\quad\quad\quad xp_{\bar{u}} \leftarrow \mathtt{F}$
R06 $\quad$ If $lt \in \text{CH}_{\bar{u}}$:
R07 $\quad\quad k \leftarrow \diamond$
R08 If $(ad, c) \notin \text{SC}_{\bar{u}}$:
R09 $\quad is_u \leftarrow \mathtt{F}$
R10 $\quad \text{CH}_{\bar{u}} \xleftarrow{\cap} [\![lt]\!]$
R11 $\text{CH}_{\bar{u}} \leftarrow \text{CH}_{\bar{u}} \setminus \{lt\}$
R12 Return k

Fig. 2.22 KuBOOM CONF game.

Theorem 2.6.1. *Let π be the KuBOOM construction in Fig. 2.20, let* AUTH *be the authenticity game in Fig. 2.2 that calls π's procedures in its oracles, let H be a perfectly collision resistant hash function, and let $\mathcal{A}$ be an adversary that makes at most q_s* Send *queries. Then there exists an adversary $\mathcal{A}'$ of comparable efficiency such that*

$$\mathbf{Adv}^{\text{AUTH}}_{\pi}(\mathcal{A}) \leq q_s \cdot \left(\mathbf{Adv}^{\text{SUF}}_{\text{OTS}}(\mathcal{A}') + \mathbf{Adv}^{\text{AUTH}}_{\text{USS}}(\mathcal{A}')\right).$$

Proof. The result immediately follows by applying Lemma 2.6.2 and Lemma 2.6.3. □

Lemma 2.6.2. *Let π be the KuBOOM construction in Fig. 2.20 and let* AUTH *be the corresponding authenticity game in Fig. 2.2 that calls π's procedures in its oracles. Let* BS *be the KuBOOM-Signature Scheme construction in Fig. 2.18 and let KuBOOM-*AUTH *be the corresponding authenticity game in Fig. 2.21. Let $\mathcal{A}$ be an adversary that makes at most q_s* Send *queries. Then there exists an adversary $\mathcal{A}'$ of comparable efficiency such that*

$$\mathbf{Adv}^{\text{AUTH}}_{\pi}(\mathcal{A}) \leq q_s \cdot \mathbf{Adv}^{\text{SUF}}_{\text{OTS}}(\mathcal{A}') + \mathbf{Adv}^{\textit{KuBOOM}\text{-AUTH}}_{\text{BS}}(\mathcal{A}').$$

Proof. $\mathcal{A}'$ will simulate the AUTH game to $\mathcal{A}$ by running π and replacing π's calls to the KuBOOM-Signature Scheme with queries to $\mathcal{A}'$'s $KuBOOM$-AUTH game. For $\mathcal{A}$'s Tick oracle queries, $\mathcal{A}'$ simply advances physical time. For each Send oracle query, $\mathcal{A}'$ will initiate a SUF game for one-time signatures [s01] and obtain a verification key vk. $\mathcal{A}'$ will pose a Sign(vk) query in the $KuBOOM$-AUTH game to obtain a signature σ_1 on vk [s02]. Finally, $\mathcal{A}'$ will pose a Sign($ad \mathbin{+\!\!+} c$) query in its SUF game to obtain a signature σ_2 on $ad \mathbin{+\!\!+} c$ [s08]. This allows $\mathcal{A}'$ to answer any Send oracle query. Notice how the Expose oracles in both games are equal, so any query can simply be forwarded by $\mathcal{A}'$. We remark $\mathcal{A}'$ does not need to know the signing keys generated by the SUF games as they are not stored in the state: They are generated, immediately used and discarded in a single send operation.

For each Recv oracle query, $\mathcal{A}'$ will first call Vfy(vk', σ_1') in the $KuBOOM$-AUTH game [r06] and subsequently Vfy$(ad' \mathbin{+\!\!+} c', \sigma_2')$ in the corresponding SUF game [r07]. If (ad', c') is a forgery and $vk = vk'$, the oracle call will award a win in the corresponding SUF game for the one-time signature. Otherwise, if $vk \neq vk'$, $\mathcal{A}$ either wins the AUTH game or renders the user out-of-sync (e.g. with a trivial forgery after exposure). Now observe $\mathcal{A}'$ has submitted vk' as message in its Vfy query in the in its $KuBOOM$-AUTH game. Hence, $\mathcal{A}'$ will also win the $KuBOOM$-AUTH game or render the user out-of-sync, maintaining equivalent game variables as the AUTH game. We can conclude $\mathbf{Adv}^{\text{AUTH}}_{\pi}(\mathcal{A}) \leq q_s \cdot \mathbf{Adv}^{\text{SUF}}_{\text{OTS}}(\mathcal{A}') + \mathbf{Adv}^{KuBOOM\text{-AUTH}}_{\text{BS}}(\mathcal{A}')$. □

Lemma 2.6.3. *Let* BS *be the KuBOOM-Signature Scheme in Fig. 2.18, let* $KuBOOM$-AUTH$_{\text{BS}}$ *be the authenticity game in Fig. 2.21 that calls* BS*'s procedures in its oracles and let* $\mathcal{A}$ *be an adversary that makes at most* q_s Sign *queries. Then there exists an adversary* $\mathcal{A}'$ *of comparable efficiency such that*

$$\mathbf{Adv}^{KuBOOM\text{-AUTH}}_{\text{BS}}(\mathcal{A}) \leq q_s \cdot \mathbf{Adv}^{\text{AUTH}}_{\text{USS}}(\mathcal{A}').$$

Proof. For simplicity let the function H in BS be the identity function. An implementation could use a collision resistant hash function for efficiency reasons. $\mathcal{A}'$ will simulate the $KuBOOM$-AUTH game to $\mathcal{A}$ and we will show that if $\mathcal{A}$ wins, then $\mathcal{A}'$ will win in one of its USS-AUTH games. Let $\mathcal{A}'$ run BS but instead of calling the gen algorithm, $\mathcal{A}'$ will initiate a new USS-AUTH for each invocation. Moreover, instead of calling the updss, updvs, sign and vfy algorithms, $\mathcal{A}'$ will query the Updss, Updvs, Sign and Vfy oracles, respectively, in the corresponding USS-AUTH game.

In particular, for each Sign oracle query by $\mathcal{A}$, $\mathcal{A}'$ starts a new USS game to generate a key pair. $\mathcal{A}'$ is free to query ExposeV to obtain the verification key and to execute

line [s02], $\mathcal{A}'$ can query the Sign oracle in its USS game. We remark BS only signs once with each key [s02,s07]. To answer a Vfy query, $\mathcal{A}'$ simply queries its Updvs (line [v10]), Vfy (line [v11]) and Updss (line [v15]) oracles in each game, as they are available without restrictions. Finally, $\mathcal{A}'$ will always be able to answer Expose queries as there are no restrictions on its ExposeS and ExposeV oracles in the USS games. It is clear the simulation always succeeds.

There are two reward conditions in *KuBOOM*-AUTH: $\mathcal{A}$ must deliver either a *certified* or an *authoritative* message/signature pair. We will first consider the reward condition for certified message/signature pairs and show that $\mathcal{A}$ cannot trigger it. The game starts without any certified message/signature pairs [G04]. A pair (m, σ) can only become certified for user u if a sync-preserving pair (m_c, σ_c) [V02] has been delivered, while u was in-sync [S01], with a higher index [V04]. Because of [S01–S02], this means $\bar{u}$ was also in-sync at the time of sending. We can conclude $\mathcal{A}$ has not interfered with (m_c, σ_c). BS will parse σ_c [v00] and obtain an array $\mathrm{S}_c[\![lt_c]\!]$ with entries for all indices $lt < lt_c$. For each lt, BS will parse $\mathrm{S}[lt]$ [v07], and store (h, σ) [v14], where $h = H(m \mathbin{+\!\!+} lt \mathbin{+\!\!+} \mathrm{P} \mathbin{+\!\!+} vk \mathbin{+\!\!+} \mathrm{S}[\![lt]\!])$ [s01]. Now, if u receives a certified pair (m', σ') it will parse it [v00] and compute $h' = H(m' \mathbin{+\!\!+} lt' \mathbin{+\!\!+} \mathrm{P}' \mathbin{+\!\!+} vk' \mathbin{+\!\!+} \mathrm{S}'[\![lt']\!])$ [v01]. Then BS requires $(h, \sigma) = (h', \sigma')$ [v20], so clearly $\mathcal{A}$ cannot forge a certified message/signature pair as the entries (h, σ) are legitimate.

Next, let us consider the reward condition for an authoritative message/signature pair. It remains to be shown that if $\mathcal{A}$ triggers this reward condition, then $\mathcal{A}'$ will win in one of its USS games. W.l.o.g. let us assume $\mathcal{A}$ submits a forgery to user u on index i that triggers the reward condition. This means u was still in-sync [V07]. Thus the verification key vk_i, which u used to verify the forgery, was generated by $\mathcal{A}'$. That is, $\mathcal{A}'$ is playing a USS game for this verification key. It is clear that if $\mathcal{A}'$ has not made an ExposeS query in the corresponding USS game, $\mathcal{A}'$ can submit the same forgery to its Vfy oracle and win in its game. Hence, let us assume $\mathcal{A}'$ has made an ExposeS query. Because BS generates a new key pair each sign invocation and overwrites the old secret key [s07], $\mathcal{A}'$ would only make this ExposeS query if $\mathcal{A}$ exposes $\bar{u}$ at sending index i. If $is_{\bar{u}}$ this implies $i \notin \mathrm{VF}_{\bar{u}}[lt]$ for all $lt < i$ and $i \notin \mathrm{AU}_{\bar{u}}$ [E00–E02]. Thus the only way to add i back to $\mathrm{AU}_{\bar{u}}$ is to deliver a message/signature pair with index $lt \geq i$ [V05]. If $\mathcal{A}$ delivers a message/signature pair with index i, $\mathcal{A}$ cannot forge on i because BS prevents double delivery [v19,v21]. If $\mathcal{A}$ delivers a pair (m, σ) with an index greater than i, we have $(m, \sigma) \in \mathrm{SC}_{\bar{u}}$, otherwise u will lose sync [V06,V10] and we know u must still be in-sync when the forgery happens [V07]. We can conclude index i becomes certified [V02–V04] and we have already shown $\mathcal{A}$ cannot forge on a certified message/signature pair.

Therefore, let us consider the final case where $\bar{u}$ is not in-sync when the exposure happens. This implies $\bar{u}$ has accepted a pair $(m, \sigma) \notin \mathrm{SC}_u$ [V06,V10]. Because u is in-sync, this means (m, σ) was not sent by u [S01–S02]. Let (m_t, σ_t) be the pair with index lt_t that transitions $\bar{u}$, i.e. renders $\bar{u}$ out-of-sync, and let $lt^*_{\bar{u}}$ be the receiving index of $\bar{u}$. Now, $\bar{u}$ will parse it [v00] and compute $h_t = H(m_t \mathbin{\#} lt_t \mathbin{\#} \mathrm{P}_t \mathbin{\#} vk_t \mathbin{\#} \mathrm{S}_t[\![lt_t]\!])$ [v01]. We have shown $\mathcal{A}$ cannot forge on certified message/signature pairs, so $lt^*_{\bar{u}} \leq lt_t$. In particular this means BS computes $av_{\bar{u}} \leftarrow H(av_{\bar{u}} \mathbin{\#} h_t \mathbin{\#} \sigma_t)$ [v13], $\mathcal{A}'$ will query $\mathrm{Updss}(av_{\bar{u}})$ in the corresponding USS game [v15] and BS includes $lt_t + 1$ in the set $\mathrm{P}_{\bar{u}}$ to indicate it has processed lt_t [v16–v17]. Let lt_f be $\bar{u}$'s sending index when going out-of-sync. We know $\bar{u}$ is not in-sync, so the next message/signature pair delivered to u with index greater than or equal to lt_f will render u out-of-sync [V06,V10]. Moreover, we know $i \geq lt_f$, because the USS game corresponding to index i was subject to an ExposeS query when $\bar{u}$ was no in-sync. Therefore this next message/signature pair delivered to u must have index i [V07]. We conclude $\mathcal{A}$ has not made an $\mathrm{Expose}(\bar{u})$ query for index lt_f while $\bar{u}$ was still in-sync, otherwise we have $i \notin \mathrm{AU}_{\bar{u}}$ [E01–E03] with no possibility of recovery.

This implies $\mathcal{A}'$ did not make an ExposeS query in the corresponding USS game before the $\mathrm{Updss}(av_{\bar{u}})$ query. Thus any string in XP contains $av_{\bar{u}}$ [E01]. When u receives the message/signature pair with index i it will iteratively update its receiving index [v02–v17]. $\mathcal{A}'$ will query the Updvs oracle with $\mathrm{AS}_u[j]$ for $j \in \mathrm{P}$ and we have $ar_{\bar{u}} \neq \mathrm{AS}_u[j]$ for all j, because we know $av_{\bar{u}} \neq \mathrm{AS}_u[lt_t]$ (and it clearly cannot match any other index as the index is part of the variable). We conclude the verifier's updates do not contain $av_{\bar{u}}$. Hence $\mathrm{AD}_V \notin \mathrm{XP}$, so $\mathcal{A}'$ would be rewarded for the forgery [V02–V03] in the USS game when querying the Vfy oracle. This completes the proof. □

2.6.2 KuBOOM achieves Confidentiality

We prove our KuBOOM construction provides confidentiality by reducing the problem to the security of the KeKEM (Sec. 2.4.3) and the KuKEM (Sec. 2.4.2). In order to do so, in Lemma 2.6.5 we first reduce the confidentiality of our KuBOOM construction to the confidentiality of the symmetric encryption scheme E, the confidentiality of our KuBOOM-KEM introduced in Sec. 2.5.2 and the authenticity of our KuBOOM construction. In Lemma 2.6.6 we reduce the advantage of an adversary against our KuBOOM-KEM in the multi-challenge confidentiality game to the advantage of an adversary in the single-challenge confidentiality game. Subsequently, we reduce the confidentiality of the KuBOOM-KEM in the single-challenge setting to the confidentiality of both the KeKEM and the KuKEM in Lemma 2.6.7.

Theorem 2.6.4. *Let π be the KuBOOM construction in Fig. 2.20, let* CONF *be the confidentiality game in Fig. 2.3 that calls π's procedures in its oracles and let $\mathcal{A}$ be an adversary that makes at most q_c* Chal *queries and ϵ the probability that the adversary successfully computes a pre-image of the random oracle. Then there exists an adversary $\mathcal{A}'$ of comparable efficiency such that*

$$\begin{aligned}\mathbf{Adv}^{\text{CONF}}_{\pi}(\mathcal{A}) \leq & q_c \cdot \left(2 \cdot \mathbf{Adv}^{\text{CONF}}_{\text{keKEM}}(\mathcal{A}') + 2 \cdot \mathbf{Adv}^{\text{CONF}}_{\text{kuKEM}}(\mathcal{A}') + \mathbf{Adv}^{\text{CONF}}_{\text{E}}(\mathcal{A}')\right) \\ & + \mathbf{Adv}^{\text{AUTH}}_{\pi}(\mathcal{A}') + \epsilon.\end{aligned}$$

Proof. The result immediately follows by successively applying Lemma 2.6.5, Lemma 2.6.6 and Lemma 2.6.7. □

Lemma 2.6.5. *Let π be the KuBOOM construction in Fig. 2.20, let* CONF^b *be the confidentiality game in Fig. 2.3 that calls π's procedures in its oracles. Let* BK *be the KuBOOM-KEM construction in Fig. 2.19 and let $KuBOOM$-*CONF^b *be the corresponding confidentiality game in Fig. 2.22. Let $\mathcal{A}$ be an adversary that makes at most q_c* Chal *queries. Then there exists an adversary $\mathcal{A}'$ of comparable efficiency such that*

$$\mathbf{Adv}^{\text{CONF}}_{\pi}(\mathcal{A}) \leq q_c \cdot \mathbf{Adv}^{\text{CONF}}_{\text{E}}(\mathcal{A}') + \mathbf{Adv}^{\text{AUTH}}_{\pi}(\mathcal{A}') + \mathbf{Adv}^{KuBOOM\text{-CONF}}_{\text{BK}}(\mathcal{A}').$$

Proof. Note the $KuBOOM$-CONF^b and the CONF^b game trace exactly the same variables. The only difference is if $\mathcal{A}$ wishes to challenge an out-of-sync user that is not *poisoned*. In this case, we have that is_u and $poisoned_u$ are both False. Note this exactly triggers the win condition in AUTH, as the game either rewards or sets *poisoned* to True when is_u becomes False. Thus, we can assume $\mathcal{A}$ only challenges an in-sync user.

$\mathcal{A}'$ will simulate the CONF^b game to $\mathcal{A}$ by running π and replacing π's calls to the $KuBOOM - KEM$ with queries to $\mathcal{A}'$s $KuBOOM$-CONF^b game. Because both games trace the same variables $\mathcal{A}'$ can simply forward all Expose, Recv, Decide queries. For any Tick queries, $\mathcal{A}'$ will query both Upd and Expire. If $\mathcal{A}$ makes a Send query, $\mathcal{A}'$ will replace the enc call in the send procedure [s04] with a query to its Enc oracle. However, for $\mathcal{A}$'s Chal queries, $\mathcal{A}'$ will replace the enc call in the send procedure [s04] with a query to its Challenge oracle. If $\mathcal{A}$ were to distinguish between $\mathcal{A}'$'s simulation and the real protocol, then $\mathcal{A}'$ would have the same distinguishing advantage in the $KuBOOM$-CONF game. Thus let us then consider the simulation variant where $\mathcal{A}'$ always use a random key. Now, for $\mathcal{A}$'s Chal queries, $\mathcal{A}'$ will initiate a standard symmetric encryption CPA game (which samples a random key) and query its encryption oracle to obtain the required ciphertext [s06]. If $\mathcal{A}$ is able to distinguish the messages based on the challenge ciphertext, $\mathcal{A}'$ can

make the same guess in the corresponding CONF game for the symmetric encryption scheme. □

Lemma 2.6.6. *Let* BK *be the KuBOOM-KEM construction in Fig. 2.19, let* $KuBOOM$-CONF^b *be the corresponding confidentiality game in Fig. 2.22 that calls* BK*'s procedures in its oracles and let* $KuBOOM$-CONF^b-1 *be the confidentiality game that is almost identical, but only allows one* Challenge *query. Let* $\mathcal{A}$ *be an adversary that makes at most* q_c Challenge *queries. Then there exists an adversary* $\mathcal{A}'$ *of comparable efficiency such that*

$$\mathbf{Adv}^{KuBOOM\text{-CONF}}_{\text{BK}}(\mathcal{A}) \leq q_c \cdot \mathbf{Adv}^{KuBOOM\text{-CONF-1}}_{\text{BK}}(\mathcal{A}').$$

Proof. Let H_i denote the hybrid of the $KuBOOM$-CONF^b game where the first i Challenge queries output the real key and the remaining challenge queries output a random key. Clearly, $H_0 = KuBOOM\text{-CONF}^1$ and $H_{q_c} = KuBOOM\text{-CONF}^0$. We define $\mathbf{Adv}^{\text{hyb}}_{i-1,i}(\mathcal{A}) \coloneqq |\mathbb{P}[H_{i-1}(\mathcal{A})] - \mathbb{P}[H_i(\mathcal{A})]|$. By the triangle inequality we have $\mathbf{Adv}^{KuBOOM\text{-CONF}}_{\text{BK}}(\mathcal{A}) \leq \sum_{i=1}^{q_c} \mathbf{Adv}^{\text{hyb}}_{i-1,i}(\mathcal{A})$. So there exists a j such that $0 < j \leq q_c$ and $\mathbf{Adv}^{KuBOOM\text{-CONF}}_{\text{BK}}(\mathcal{A}) \leq q_c \cdot \mathbf{Adv}^{\text{hyb}}_{j-1,j}(\mathcal{A})$. It remains to be shown that we can use an adversary $\mathcal{A}$ that can distinguish between H_{j-1} and H_j to win the $KuBOOM$-CONF^b-1 game.

$\mathcal{A}'$ will initialize the $KuBOOM$-CONF^b-1 and run $\mathcal{A}$. Any query by $\mathcal{A}$ to the Expose, Enc, Dec, Upd or Expire, $\mathcal{A}'$ can simply forward to its own oracles but will keep track of the game variables. When $\mathcal{A}$ makes a Challenge(u, ad) query, $\mathcal{A}'$ will first check if it is a valid challenge. If the requirements are not met, $\mathcal{A}'$ can abort as $\mathcal{A}$ would lose the game anyway. Otherwise $\mathcal{A}'$ will proceed as follows, where t counts the number of Challenge queries. If $t < j$, $\mathcal{A}'$ makes the corresponding Enc(u, ad) query, if $is_{\bar{u}}$ will add lt_u to CH_u and return (k, c) to $\mathcal{A}$. If $t = j$, $\mathcal{A}'$ will forward the query to the Challenge oracle in its own game. If $t > j$, $\mathcal{A}'$ makes the corresponding Enc(u, ad) query, if $is_{\bar{u}}$ will add lt_u to CH_u, replace $k \leftarrow_{\$} \mathcal{K}$ and return (k, c) to $\mathcal{A}$.

Now observe $\mathcal{A}'$ simulates H_{j-1} to $\mathcal{A}$ if $\mathcal{A}'$ is playing the $KuBOOM$-CONF^1-1 game and H_j if $\mathcal{A}'$ is playing $KuBOOM$-CONF^0-1. Thus, when $\mathcal{A}$ makes its guess, $\mathcal{A}'$ can make the corresponding guess in its own game. We conclude $\mathbf{Adv}^{\text{hyb}}_{j-1,j}(\mathcal{A}) \leq \mathbf{Adv}^{KuBOOM\text{-CONF-1}}_{\text{BK}}(\mathcal{A})$. □

Lemma 2.6.7. *Let* BK *be the KuBOOM-KEM construction in Fig. 2.19, let* $KuBOOM$-CONF^b *be the corresponding confidentiality game in Fig. 2.22 that calls* BK*'s procedures in its oracles but only allows one challenge query. Let* $\mathcal{A}$ *be an adversary.*

Then there exists an adversary $\mathcal{A}'$ of comparable efficiency such that

$$\mathbf{Adv}^{KuBOOM\text{-CONF-1}}_{\mathrm{BK}}(\mathcal{A}) \leq 2 \cdot \left(\mathbf{Adv}^{\mathrm{CONF}}_{\mathrm{keKEM}}(\mathcal{A}') + \mathbf{Adv}^{\mathrm{CONF}}_{\mathrm{kuKEM}}(\mathcal{A}')\right).$$

Proof. To prove the result we will show we can use an adversary that has a distinguishing advantage in the $KuBOOM$-CONF^b game has at least half that advantage in the keKEM-CONF^b or the kuKEM-CONF^b game. Let $\mathcal{A}'$ simulate the $KuBOOM$-CONF game by running BK. However, instead of calling the gen algorithms, it will initialize KeKEM and KuKEM confidentiality games. Moreover, $\mathcal{A}'$ will replace any call to these primitives' algorithms with queries to the corresponding oracles in its confidentiality games. That is enc, dec, evolveps, evolvess and expire for the KeKEM primitive and enc, dec, updps and updss for the KuKEM primitive.

To answer $\mathcal{A}$'s Challenge(u, ad) query, $\mathcal{A}'$ will check if is_u and abort otherwise as $\mathcal{A}$ would lose the $KuBOOM$-CONF game [C00]. This implies the latest public states u received were indeed generated by $\mathcal{A}'$.

Let us first consider the case $is_{\bar{u}} = \mathtt{T}$. $\mathcal{A}'$ will guess uniformly at random whether $\mathcal{A}$ will let the challenge ciphertext expire or deliver it first. Then $\mathcal{A}'$ can query the Challenge oracle in the corresponding KeKEM or KuKEM game, depending on its guess, and run the enc procedure for the other primitive to execute the lines [e00,e01] in BK. Because BK always uses the latest received public states [d07,d10] for encapsulation, we can conclude from [R04,R05,E02,E03] that flag $xp_{\bar{u}}$ reflects whether the secret states corresponding to the current public states used for encapsulation have been exposed. If $xp_{\bar{u}}$, then $\mathcal{A}'$ can abort as $\mathcal{A}$ would lose anyway [C01]. Thus, we can conclude $\mathcal{A}'$ has not had to query its Expose oracles in the current games and the Challenge query will succeed.

Next, let us consider the case $is_{\bar{u}} = \mathtt{F}$. Now, $\mathcal{A}'$ will always query the Challenge oracle in its KuKEM game. If an Expose query was made before $\bar{u}$ went out-of-sync, analogously to above, the flag $xp_{\bar{u}}$ will be set and $\mathcal{A}'$ can abort the Challenge query. Hence, assume only Expose queries were made after $\bar{u}$ went out-of-sync. We know $\mathcal{A}'$ updated the KuKEM states when running the dec procedure [d22], updating AD_R in its KuKEM game. By [E10,E12] in the KuKEM CONF game, we know all exposed strings are prefixed with AD_R. We can conclude $\mathrm{AD}_S \notin \mathrm{XP}$ and the Challenge query will succeed [C00].

Now, let us discuss how to answer Expose queries. It is clear if there has been no Challenge query that $\mathcal{A}'$ can make the required ExposeR in its games to answer $\mathcal{A}$'s Expose queries. So, let us assume $\mathcal{A}$ has made a Challenge query. We first remark if $\bar{u}$ was out-of-sync at the moment of the Challenge query, $\mathcal{A}'$ will have challenged the

KuKEM game and since $\text{AD}_S \notin \text{FE}_R$, $\mathcal{A}'$ is free to query ExposeR in the KuKEM game [E11]. Clearly, in this scenario $\mathcal{A}'$ can also make an ExposeR query in the KeKEM game, so let us assume $\bar{u}$ was in-sync and the challenge ciphertext has been added to the challenge set CH_u [C05]. By requirement [E00], $\mathcal{A}$ will not make an Expose query until either the challenge ciphertext expired or the challenge ciphertext is delivered / goes out-of-sync. Suppose $\mathcal{A}$ calls the Tick oracle to expire the challenge ciphertext. Now, $\mathcal{A}'$ will have called the Expire oracle in the KeKEM game [x02], which allows $\mathcal{A}'$ to make Expose queries again [X01] in the KeKEM game and if $\mathcal{A}'$ guessed correctly it did not make a Challenge query in the KuKEM game, so it is free to call the Expose oracle.

Now suppose, $\mathcal{A}'$ clears challenge set CH by delivering a ciphertext. By [R10,R11] this means $\mathcal{A}$ has delivered the challenge ciphertext or an out-of-sync ciphertext with index less or equal than the challenge ciphertext. In the either case, the KuKEM will update [d22] and $\mathcal{A}'$ can query ExposeR again because $\text{AD}_S \notin \text{FE}_\text{R}$ [E10].

Finally, $\mathcal{A}'$ will call K to compute the combined key [e09]. For simplicity we will assume K to be a random oracle. We remark if $(ad, c) \in \text{SC}$, $\mathcal{A}'$ can simply answer $\text{Dec}(\bar{u}, ad, c)$ queries for challenge ciphertexts with $k \leftarrow \diamond$ by [R06] and otherwise we can sample a $k \leftarrow_\$ \mathcal{K}$ because random oracle K takes the full (ad, c) as context [d25], as long as we maintain consistency for repetitive Dec queries. Because K is a random oracle, if $\mathcal{A}$ can distinguish which $KuBOOM\text{-CONF}^b$ game it is playing, it must have learnt the challenged key as it is input to K. (Note that if $\mathcal{A}$ guesses the key, $\mathcal{A}'$ will have at least the same success probability guessing the key in its own game.) Therefore, $\mathcal{A}'$ will have the same advantage in either its KeKEM or its KuKEM game. We remark if $\mathcal{A}'$ guesses correctly, the simulation always succeeds. $\mathcal{A}$'s view is independent of $\mathcal{A}'$'s guess, so the probability of a correct guess is at least $\frac{1}{2}$. □

2.7 Conclusion and Reflection

After ACD [5] observed that research on secure messaging protocols routinely only considers settings with a guaranteed in-order delivery of messages, while most real-world protocols like Signal are actually designed for out-of-order delivery, we reassess the model and construction of ACD and argue that the intuitive notion of forward secrecy is not provided. We identify that the reason for this is the lack of modelling of physical time, which is required to express that ciphertexts may time out and expire. We hence develop new security models for the out-of-order delivery setting with immediate decryption. Our model incorporates the concept of physical clocks and implements a maximally strong corruption model. We finally design a proof-of-concept protocol that provably satisfies it.

We would like to stress that our security model for secure messaging cannot be reduced to merely the addition of physical time and the deletion of keys and discuss comparisons with the Signal protocol and variants thereof. Our model should be considered in conjunction with the other security guarantees, which explains why it is not sufficient to simply tweak the Signal protocol to "forget old keys". In fact, regardless of Bob's expiration policy a clock-augmented Signal, falls prey to the following attack: Expose Alice's state, let her encrypt/send a challenge message, use the obtained state to decrypt the ciphertext. While that seems an unfair attack on Signal's use of symmetric encryption, the point is that in the PKE-strengthened Signal variant discussed in ACD19, the situation immediately becomes less clear: Keys cannot be deleted without destroying the capability of further communication as each party only holds one private key for subsequent messages.

Observe however that our model only excludes trivial attacks, as listed in Sec. 2.3. In particular, the adversary is allowed to freely expose Bob without consequences in our model when Bob is out-of-sync. To be concrete, consider the following attack which shows that a PKE-strengthened Signal is insecure in our model, even if keys would be deleted. The adversary exposes Alice's state and uses it to simulate the consecutive encryption of two messages. The adversary obtains two ciphertexts, discards the first one and delivers the second one to Bob. Then the adversary exposes Bob's state and, finally, lets Alice encrypt the challenge message that it wants to break. (This would be the first message she processes, i.e., it is for sending index 0.) In the Signal protocol, the adversary could readily recover the challenge message by invoking the decryption algorithm with the state priorly revealed from Bob. However, intuitively, and according to our models and as ensured by our construction, the challenge message should remain confidential. Note that "forgetting old keys" wouldn't resolve anything here as from Bob's view, nothing is "old": time has not ticked.

In Sec. 2.3 we provided an attack catalogue that lists over ten *non-trivial* attacks, similar to the one described above, that the Signal protocol falls prey to. This clearly shows the Signal protocol is insecure in our models and justifies the more expensive primitives used. In our work we have defined what optimal security is in the immediate decryption setting and provided a proof of concept to demonstrate it can be achieved. We do believe that protocols less efficient but more secure than Signal have their merit and it is up to the user to make this security / efficiency trade-off.

We refer to our models as 'optimally secure' because they exclude only trivial attacks (in a very fair communication and corruption model; of course our models limit the adversary to specific actions through oracles, but the latter is trivially true for *any*

security model). However, we remark there has been discussion in the field of the term 'optimal' in the case of state reveal queries over randomness reveal queries. We chose state reveal (SR) over randomness reveal (RR) for a number of reasons. First, state reveal attacks are very much of a concern to practitioners. Second, if a construction shall be proven secure in a model with randomness reveal, then necessarily all its building blocks need to be RR-secure as well. For instance, if a construction uses a generic signature scheme as a building block (simplest example: Signed-DH), then the latter has to be RR-unforgeable (Existential unforgeability in a model with signer randomness reveal). The fewest practical signature schemes are formally analysed in such a model, and many Fiat-Shamir based schemes (including DSA, ECDSA) are non-candidates, as for them revealing the signing randomness is equivalent with leaking the key. Deterministic signature schemes could be used, but they are more rare and inherently less secure (considerable tightness losses [10]). As another example, if a construction is based on a PKE primitive, then the latter has to be RR-confidential (e.g., CCA with encryptor RR resilience); such primitives have been proposed under the umbrella of selective opening security [16], but corresponding constructions are computationally expensive. Effectively, a choice of randomness reveal over state reveal would rule out many otherwise secure constructions for no reason that we believe warrants it. Observing that, in absolute numbers, mobile devices are overwhelmingly idle and that exposures due to physical events are unlikely to happen *during* instant messaging operations, we believe randomness reveal additional value is limited: The adversary can already reveal the state of Alice before and after an operation.

We note that optimally secure protocols are in general more costly than weaker versions. Our solution, in contrast to Signal, seems to require HIBE which is admittedly one of the more expensive primitives. For future work it would be interesting to have a prototype implementation to benchmark how much one is 'paying' for this additional security compared to the Signal protocol. However, if a practitioner desires to avoid using HIBE at all cost (for efficiency, and accepting the implied security loss), it is straightforward to replace our KuKEM building block with a more efficient KEM+Hash construction that does not achieve optimal security. For reference we provide the details of such a construction in Fig. 2.23. If our messaging protocol is instantiated with this alternative building block, rather than with our (HIBE-based) KuKEM, the protocol would still be more secure than Signal: It would fall prey to some of the attacks listed in Sec. 2.3, but for example not to the trivial attack listed in item 4 of the confidentiality attack catalogue (as Signal would do).

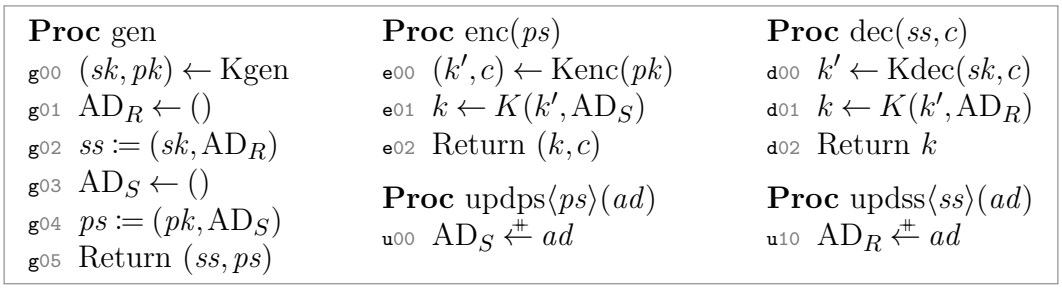

Fig. 2.23 Insecure KuKEM construction. Building block is a KEM (Kgen, Kenc, Kdec) and key derivation function K.

A further contentious point is the assumption of loosely synchronized clocks in an adversarial setting. Desynchronized clocks would harm availability: By desynchronizing Alice and Bob, the adversary can achieve that they cannot communicate any further. However, this is similar to the adversary cutting the wire between them. No protocol —in no model— can realistically defend against this. As highlighted in Footnote 6, with today's infrastructure users will not desynchronize by considerable amounts by accident. We intentionally don't prescribe a specific expiration duration δ, as different users will have different needs/preferences. Highly exposed users might choose $\delta = 5$ mins; other users might configure $\delta = 1$ month. Our solutions are flexible and work with any δ.

While we don't know about the reader's personal phones, the author can report that we did not notice a desynchronization of our phones from 'real time' by more than one minute, ever having taken place in the past ten years. To us, it is virtually unimaginable to believe that a noticeable share of users runs around with considerably wrongly-set mobile phone clocks. Again, as desynchronization at most harms availability, even picking $\delta = 15$ mins seems to be an extremely robust choice in practice. For practicality users may wish to select a much larger δ than 15 mins though. If the phone is offline, for example on a flight, then it will still expire the keys after the specified period δ for security. Once the phone comes back online and receives new messages, the messages whose keys have expired cannot be decrypted. Of course the application's user interface can still display to the user that an old undecipherable message was received, so the user will be aware they missed parts of the conversation.

Finally, we remark that the ciphertext size of our proposed construction grows linearly in the number of *unacknowledged* messages. Since ciphertexts are acknowledged with each message, the ciphertext size only grows linearly in the number of messages sent since the last received message, it is *not* linear in the number of sent messages. Recall the discussion in Sec. 2.5.2 shows it is indeed necessary that ciphertexts contain information about prior undelivered ciphertexts to achieve our security notion, so a constant size

ciphertext construction is impossible in our security model. However, we consider this a feature. Effectively, it automatically implements graceful rate limiting if someone keeps spamming you and you ignore them, by slowly increasing their workload every message.

Chapter 3

Encrypt-to-self: Securely Outsourcing Storage

We explore techniques that enable a computing device to securely outsource the storage of data. The setting assumes a memory-bounded computing device that inflates the amount of volatile or permanent memory available to it by letting other (untrusted) devices hold encryptions of information that they return on request. In this chapter we develop the cryptographic mechanism that should be used to achieve confidential and authentic data storage in the encrypt-to-self setting, i.e., where encryptor and decryptor coincide and constitute the only entity holding keys. We argue that standard authenticated encryption represents only a suboptimal solution for preserving confidentiality, as much as message authentication codes are suboptimal for preserving authenticity. The crucial observation is that such schemes instantaneously give up on *all* security promises in the moment the key is compromised. In contrast, data protected with our new primitive remains fully integrity protected and unmalleable. In the course of this chapter we develop a formal model for encrypt-to-self systems, show that it solves the outsourced storage problem, propose surprisingly efficient provably secure constructions, and report on our implementations.

3.1 Introduction

We start with motivating this area of research by describing four application scenarios where outsourcing storage might prove crucial.

Web Server. Web servers typically hold a multitude of data for each of the client connections they manage, ranging from user preferences to technical information like

database credentials. If the amount of data per session is considerable, busy servers sooner or later run out of memory. One admissible solution to this is to let the server *encrypt* the session data *to itself* and to let the client store the ciphertext, with the agreement that the client reproduce the ciphertext in each subsequent request (e.g., via a cookie) so that the session data can be recovered when required. While it is difficult to make general statements about the setup of a web server back-end, it is fair to say that the processing of HTTP requests routinely also includes extracting a session identifier from the HTTP header and fetching basic session-related information (e.g., the user's password, the date and time of the last login, the number of failed login attempts, but also other kinds of data not related to security) from a possibly remote SQL database. To avoid the inherent bottleneck induced by the transmission and processing of the database query, such data can be cached on the web server, the limits of this depending only on the amount of available working memory (RAM). For some types of web applications and a large number of web sessions served simultaneously, these memory-imposed limits might represent a serious restriction to efficiency. This chapter scouts techniques that allow the web server to securely outsource the storage of session information to the (untrusted) web clients. Note that we will introduce a one-time primitive, meaning each key may be used for at most one encryption. However, a web server could derive a fresh key for each encryption from its master key and session identifiers.

Hardware Security Module. An HSM is a computing device that performs cryptographic and other security-related operations on behalf of the owning user. While such devices are internally built from off-the-shelf CPUs and memory chips, a key concept of HSMs is that they are specially encapsulated to protect them against physical attacks, including various kinds of side channel analysis. One consequence of this tamper-proof shielding is that the memory capacity of an HSM can never be physically extended—unlike it would be the case for desktop computers—so that the amount of available working memory might constitute a relevant obstacle when the HSM is deployed in applications with requirements that increase over time (e.g., due to a growing user base). This chapter scouts techniques that allow the HSM to securely outsource the storage of any kind of valuable information to the (untrusted) embedding host system.

Smartcard. A smartcard, most prominently recognized in the form of a payment card or a mobile phone security token, is effectively a tiny computing device. While fairly potent configurations exist (with 32-bit CPUs and a couple of 100KBs of memory), as the costs associated with producing a smartcard scales roughly linearly with the amount of implemented physical memory, in order to be cost effective, mass-produced cards tend to come with only a small amount of memory. This chapter scouts techniques that allow

smartcards to securely outsource the storage of valuable information to the infrastructure they connect to, e.g., a banking or mobile phone backbone, or a smartphone.

Trusted Platform Module. A TPM is a discrete security chip that is embedded into virtually all laptops and desktop PCs produced in the past decade. A TPM supports its host system by offering trusted cryptographic services and is typically relied upon by boot loaders and operating systems. TPMs are located conceptually between HSMs and smartcards, and as much as these they benefit from a secure option to outsource storage.

3.1.1 Contributions

Outsourced Storage based on Symmetric Cryptography

If a computing device has access to some kind of external storage facility (a memory chip wired to it, a connected hard drive, cloud storage, etc.), then, intuitively, it can virtually extend the amount of memory available to it by outsourcing storage, i.e., by serializing data objects and communicating them to the storage facility which will reproduce them on request. In this chapter we focus on the case where neither the external storage facility nor the connection to it is considered trustworthy. More concretely, we assume that all infrastructure outside of the computing device itself is under control of an adversary that aims at reading or changing the data that is to be externally stored.[1] As a first approximation one might conclude that standard tools from the domain of symmetric encryption are sufficient to achieve security in this setting. Consider for instance the following approach based on authenticated encryption (AE, [87]): The computing device samples a fresh symmetric key; whenever it wants to store internal data on the outsourced storage, it encrypts and authenticates the data by invoking the AE encryption algorithm with its key and hands the resulting ciphertext over to the storage facility; to retrieve the data, it requests a copy of the ciphertext, and decrypts and verifies it. While this simple solution requires further tweaking to thwart replay attacks,[2] as long as the AE key remains private it can be used to protect confidentiality and integrity as expected.

[1] Certainly, the storage device can always decide to "fail" by not returning any data previously stored into it, leading to an attack on the *availability* of the computing device. We hence consider environments where this is either not a problem or where such an attack cannot be prevented anyway (independently of the storage technique). Note that this assumption holds for our four motivating scenarios.

[2] One option to strengthen the scheme against replay is to implement the AE primitive nonce-based [88], and using a strictly increasing nonce for encrypting and decrypting.

Our Contribution: Secure Outsourced Storage with Key Leakage

While we confirm that standard cryptographic methods will securely solve the storage outsourcing problem if the used key material remains private, we argue that satisfactory solutions should go a step further by providing as much security as possible even if the latter assumption (that keys remain private) is not met. Indeed, different attacks against practical systems that lead to partial or full memory leakage continue to regularly emerge (including different types of side channel analysis against embedded systems,[3] cold-boot attacks against memory chips,[4] Meltdown/Spectre-like attacks against modern CPUs,[5,6] etc.), and it is commonly understood that the corruption model considered for cryptographic primitives should always be as strong as possible and affordable. For two-party symmetric encryption (e.g., AE) this strongest model necessarily excludes any type of user corruption[7] as the keys of both parties are identical: Once any party is corrupted, any past or future ciphertext can be decrypted and ciphertexts can be forged for any message, i.e., no form of confidentiality or authenticity remains. We point out, however, that for outsourced storage a stronger corruption model is both feasible and preferable. Clearly, like in the AE case, if the adversary obtains a copy of the used key material then all confidentiality guarantees are lost (the adversary can decrypt what the device can decrypt, that is, everything), but a similar reasoning with respect to integrity protection cannot be made. To see this, consider the encrypt-then-hash (EtH) solution where the computing device encrypts the outsourced data as described above, but in addition to having the ciphertext stored externally it internally registers a hash of it (computed with, say, SHA256). When the device decides to recover externally stored data, it requests a copy of the ciphertext, recomputes its hash value, and decrypts only if the hash value is consistent with the internally registered value. Note that even if the device is corrupted and its keys became public, all successfully decrypted ciphertexts are necessarily authentic.

The example just given shows that while no solution for secure storage outsourcing can do much about protecting data confidentiality against key leakage attacks, solutions can *fully* protect the integrity of the stored data in any case. Naive AE-based schemes do not provide this type of security, and the contribution of our work is to fill this gap and to explore corresponding constructions. Precisely, we provide the following: (1) We identify the new *encrypt-to-self* (ETS) primitive as the right cryptographic tool to solve

[3] https://en.wikipedia.org/wiki/Side-channel_attack
[4] https://en.wikipedia.org/wiki/Cold_boot_attack
[5] https://en.wikipedia.org/wiki/Meltdown_(security_vulnerability)
[6] https://en.wikipedia.org/wiki/Spectre_(security_vulnerability)
[7] We use the terms 'key leakage', 'user corruption', and 'state corruption' synonymously.

the outsourced storage problem and formalize its syntax and security properties. (2) We formalize notions more directly related to the outsourced storage problem and provably confirm that secure solutions based on ETS are indeed immediate. (3) We design provably secure constructions of ETS from established cryptographic primitives.[8] (4) We develop open-source implementations of our constructions that are optimized with respect to security and efficiency.

3.1.2 Related Work

While we are not aware of any former systematic treatment of the encrypt-to-self (ETS) primitive, a number of similar primitives or ad hoc constructions partially overlap with our results. We discuss these in the following, but emphasize that none of them provides general solutions to the ETS problem.

Memory Encryption in Modern CPUs. Recent desktop and server CPUs offer dedicated infrastructure for memory encryption,[9] with the main applications in cloud computing and Trusted Execution Environments (TEEs). Prominent TEE examples include Intel SGX[10] and ARM TrustZone[11] in which every memory access of the processes that are executed within a TEE (aka 'enclave') is conducted through a memory encryption engine (MEE). This effectively implements outsourced data storage, but with quite different access rules and patterns than in the ETS case. While we consider the (stateless) encryption of a message to a ciphertext and then a decryption of a ciphertext back to a message, MEEs are stateful systems that consider the protected physical memory area a single ciphertext that is constantly locally modified with each write operation [54].

Password Managers. A password manager can be seen as a database that stores security credentials in an encrypted form and requires e.g., a master password to be unlocked. Also this can be seen as an ETS instance, but the cryptographic design of password managers has a different focus than general outsourced storage. More concretely, the central challenge solved by good password managers is the password-based key derivation,[12] which typically involves invoking a time-expensive derivation function like PBKDF2 [67] or a memory-hard derivation function like ARGON2 [23]. Password-based key derivation is not considered in our treatment of the ETS primitive

[8] The above encrypt-then-hash (EtH) solution is secure in our models but requires two passes over the data. Our solutions are more efficient, getting along with just one pass.

[9] https://software.intel.com/en-us/blogs/2017/12/22/intel-releases-new-technology-specification-for-memory-encryption

[10] https://software.intel.com/en-us/sgx/details

[11] https://genode.org/documentation/articles/trustzone

[12] https://1password.com/files/1Password-White-Paper.pdf

(we instead assume uniform keys). We refer the reader to Chapter 4 for our proposed methods to obtain uniform keys from passwords.

Encryptment. A symmetric encryption option that recently emerged as a proposal to protect messages in instant messaging is Encryptment [45]. Its features go beyond regular authenticated encryption in that the tags contained in ciphertexts act as (cryptographically strongly binding) commitments to the encoded messages. This committing feature was deemed helpful for the public resolution of cyber harassment cases by allowing affected parties to appeal to a judging authority by opening their ciphertexts by releasing their keys. On first sight this has nothing to do with our ETS setting (in which only one party holds a key, this key would never be deliberately shared, and a necessity of provably releasing message contents to anybody else is not considered). Interestingly, however, our constructions of ETS are very similar to those of [45]. The intuitive reason for this is that the ETS setting requires that ciphertexts remain unforgeable under key leakage, which somewhat aligns with the committing property of encryptment that is required to survive disclosing keys to a judge. Ultimately, however, the applications and thus security models of ETS and encryptment differ, and our constructions are actually more efficient than those in [45].[13]

3.1.3 Technical Approach.

In addition to formalizing the security of the encrypt-to-self (ETS) primitive, in the course of this chapter we also propose efficient provably-secure constructions from standardized building blocks. As discussed above, the authenticity promises of ETS shall withstand adversaries that have knowledge of the key material. In this setting one cannot hope that standard secret-key authentication building blocks like MACs or universal hash functions will be of help, as generically they lose all security when the key is leaked. We instead employ, as they manifest *unkeyed* authentication primitives, cryptographic hash functions like SHA256. A first candidate construction, already hinted at above, would be the encrypt-then-hash (EtH) approach where the message is first encrypted (using any secret key scheme, e.g., AES-CTR) and the ciphertext is then hashed. Our constructions are more efficient than this by exploiting the structure of Merkle–Damgård (MD) hash functions and dual-use leveraging on the properties of their inner building block: the compression function (CF). Intuitively, for authentication we build on the collision resistance of the CF, and for confidentiality we build on a PRF-like property of

[13]This is the case for at least two reasons: (1) The ETS primitive does not need to be committing to the key, which is the case for encryptment. (2) Our message padding is more sophisticated than that of [45] and does not require the processing of a length field.

the CF. More precisely, our message schedule for the CF is such that each invocation provides both confidentiality *and* integrity for the processed block. This effectively halves the computational costs in comparison to the EtH approach.

Since this chapter focuses on efficiency improvements over the naive EtH solution, we believe the analysis is not complete without also implementing the construction under consideration. This is because only implementing a scheme will enforce making conscious decisions about all its details and building blocks, and these decisions may crucially affect the obtained security and efficiency. We thus realized three ready-to-use instances of the ETS primitive, based on the CFs of the top performing hash functions SHA256, SHA512, and BLAKE2. In fact, observations from implementing the schemes led to considerable feedback to the theoretical design which was updated correspondingly. One example for this is connected to memory alignment: Computations on modern CPUs experience noticeable efficiency penalties if memory accesses are not aligned to specific boundaries. Our constructions reflect this at two different levels: at the register level and at the cache level (64 bit alignment for register-oriented operations, and 256 bit alignment for bulk memory transfers[14]).

3.2 Preliminaries

3.2.1 Memory Alignment

For n a power of 2, we say an address of computer memory is n-byte aligned if it is a multiple of n bytes. We further say that a piece of data is n-byte aligned if the address of its first byte is n-byte aligned. A modern CPU accesses a single (aligned) word in memory at a time. Therefore, the CPU performs reads and writes to memory most efficiently when the data is aligned. For example, on a 64-bit machine, 8 bytes of data can be read or written with a single memory access if the first byte lies on an 8-byte boundary. However, if the data does not lie within one word in memory, the processor would need to access two memory words, which is considerably less efficient. Our scheme algorithms are designed such that when they need to move around data, they exclusively do this for aligned addresses. In practice, the preferred alignment value depends on the hardware used, so for generality in this thesis we refer to it abstractly as the memory alignment value mav. (A typical value would be mav = 8.)

[14]The value 256 stems from the size of the cache lines of 1st level cache.

3.2.2 Tweaking the Compression Functions of Hash Functions

The main NIST hash functions of the SHA2 family (FIPS 180-4, [80]) accomplish their task of hashing a message into a short string by strictly following the Merkle–Damgård framework: All inputs to their core building block —the compression function— are either directly taken from the message or from the chaining state. It has been recognized, however, that options to further contextualize or domain-separate the inputs of compression functions can be of advantage. Indeed, compression functions that are designed according to the alternative, more recent HAIFA framework [22] have a number of additional inputs, for instance an explicit salt input, that allow for weaving some extra bits of context information into the bulk hash operations. A concrete example for this is the compression function of the popular BLAKE2 hash function ([9, 89], a HAIFA design), which takes as an additional input a Boolean finalization flag that is to be set specifically when processing the very last (padded) block of a hash computation. The idea behind making the last invocation "special" is that this effectively thwarts length extension attacks: While conducting extension attacks against the SHA2 hash functions, where the compression functions do not natively support any such marking mechanism, is quite immediate,[15] similar attacks against BLAKE2 are impossible [39]. We note that, generally speaking, an ad hoc way of augmenting the input of a compression function by an additional small number of bits is to XOR predefined constants into the hashing state (e.g., before or while the compression function is executed), with the choice of constants depending on the added bits. For instance, if the finalization flag is set, the BLAKE2 compression function will flip all bits of one of its inputs, but beyond that operate as normal.

While textbook SHA2 does not support contextualizing compression function invocations via additional inputs, we observe that NIST, in order to solve an emerging domain-separation problem in the definition of their FIPS 180-4 standard, employed ad hoc modifications of some SHA2 functions that can be seen as (implicitly) retrofitting a one-bit additional input into the compression function. Concretely, the SHA512/t functions [80], that intuitively represent plain SHA512 truncated to $0 < t < 512$ bits, are carefully designed such that for any $t_1 \neq t_2$ the functions SHA512/t_1 and SHA512/t_2 are independent of each other.[16] The separation of the individual SHA512/t versions works as follows [80, Sec. 5.3.6]: First compute the SHA512 hash value of the string `"SHA512/xxx"` (where placeholder `xxx` is replaced by the decimal encoding of t), then

[15] For instance, an adversary who does not know a value x but instead the values $H(x)$ and y, can compute $H(x \mathbin{\|} y)$ by just continuing the iterative MD computation from chain value $H(x)$ on. Note this does not require inverting the compression function.

[16] In particular, for instance, SHA512/128(`"a"`) is not a prefix of SHA512/192(`"a"`).

XOR the byte value `0xa5` (binary: `0b10100101`) into every byte of the resulting chain state, then continue with regular SHA512 steps from that state on, truncating the final hash value to t bits. While the XORing step is ad hoc, it arguably represents a fairly robust domain separation method for SHA2.

Our constructions of the encrypt-to-self primitive rely on compression functions that are *tweaked* with a single bit, that is, that support one bit as an additional input. When we implement this based on SHA2 compression functions, we employ precisely the mechanism scouted by NIST: When the additional tweak bit is set, we XOR constant `0xa5` into all state bytes and continue operation as normal. Our BLAKE2 based construction, on the other hand, uses the already existing finalization bit.

3.3 Foundations of Encrypt-to-Self

The overall goal of this chapter is to provide a secure solution for outsourced storage. We identified the novel encrypt-to-self (ETS) primitive, which provides one-time secure encryption with authenticity guarantees that hold beyond key compromise, as the right tool to construct outsourced storage.[17] In this section we first formalize and study ETS, then formalize outsourced storage, and finally show how the former immediately implies the latter. This allows us to leave the outsourced storage topic aside in the remaining part of the chapter and lets us instead fully focus on constructing and implementing ETS.

3.3.1 Syntax and Security of ETS

ETS consists of an encryption and a decryption algorithm, where the former translates a message to a *binding tag* and a ciphertext, and the latter recovers the message from the tag-ciphertext pair. For versatility the two operations further support the processing of an associated-data input [87] which has to be identical for a successful decryption.

The task of the binding tag is to prevent forgery attacks: A user that holds an authentic copy of the binding tag will never accept any ciphertext they did not generate themselves, even if all their secrets become public. Note that while standard authenticated encryption (AE) does not provide this type of authentication, the encrypt-then-hash construction suggested in Sec. 3.1 does. In Sec. 3.4 we provide a considerably more efficient construction that uses a hash function's compression function as its core building block. Here, we define the generic syntax of ETS and formalize its security requirements.

[17] While ETS is novel, note that prior work explored the quite similar Encryptment primitive [45]. Encryptment is stronger than ETS, and less efficient to construct.

Game FUNC$(ad, m, \mathcal{A})$
00 $k \leftarrow_{\$} \mathcal{K}$
01 $(bt, c) \leftarrow \text{enc}(k, ad, m)$
02 Invoke $\mathcal{A}(k, ad, m, bt, c)$
03 Lose

Oracle Dec(ad', c')
04 $m' \leftarrow \text{dec}(k, bt, ad', c')$
05 If $(ad', c') = (ad, c)$:
06 Promise $m' = m$
07 $m' \leftarrow \perp$
08 Return m'

Game INT$(ad, m, \mathcal{A})$
09 $k \leftarrow_{\$} \mathcal{K}$
10 $(bt, c) \leftarrow \text{enc}(k, ad, m)$
11 Invoke $\mathcal{A}(k, ad, m, bt, c)$
12 Lose

Oracle Dec(ad', c')
13 $m' \leftarrow \text{dec}(k, bt, ad', c')$
14 Reward $(ad', c') \neq (ad, c)$
15 $m' \leftarrow \perp$
16 Return m'

Game IND$^b(ad, m^0, m^1, \mathcal{A})$
17 $k \leftarrow_{\$} \mathcal{K}$
18 Require $m^0 \equiv m^1$
19 $(bt, c) \leftarrow \text{enc}(k, ad, m^b)$
20 $b' \leftarrow \mathcal{A}(ad, m^0, m^1, bt, c)$
21 Stop with b'

Oracle Dec(ad', c')
22 $m' \leftarrow \text{dec}(k, bt, ad', c')$
23 If $(ad', c') = (ad, c)$:
24 $m' \leftarrow \perp$
25 Return m'

Fig. 3.1 Games for ETS. For the values ad', c' provided by the adversary we require that $ad' \in \mathcal{AD}, c' \in \mathcal{C}$. Assuming $\perp \notin \mathcal{M}$, we encode suppressed messages with $\perp$. For the meaning of instructions Stop with, Lose, Promise, Reward, and Require see Sec. 1.3.

Definition 3.3.1. *Let $\mathcal{AD}$ be an associated data space and let $\mathcal{M}$ be a message space. An encrypt-to-self* (ETS) *scheme for $\mathcal{AD}$ and $\mathcal{M}$ consists of algorithms* enc, dec*, a key space $\mathcal{K}$, a binding-tag space $\mathcal{B}t$, and a ciphertext space $\mathcal{C}$. The encryption algorithm* enc *takes a key $k \in \mathcal{K}$, associated data $ad \in \mathcal{AD}$ and a message $m \in \mathcal{M}$, and returns a binding tag $bt \in \mathcal{B}t$ and a ciphertext $c \in \mathcal{C}$. The decryption algorithm* dec *takes a key $k \in \mathcal{K}$, a binding tag $bt \in \mathcal{B}t$, associated data $ad \in \mathcal{AD}$ and a ciphertext $c \in \mathcal{C}$, and returns a message $m \in \mathcal{M}$. A shortcut notation for this API is*

$$\mathcal{K} \times \mathcal{AD} \times \mathcal{M} \to \text{enc} \to \mathcal{B}t \times \mathcal{C} \qquad \mathcal{K} \times \mathcal{B}t \times \mathcal{AD} \times \mathcal{C} \to \text{dec} \to \mathcal{M} \ .$$

Correctness and Security. We require of an ETS scheme that if a message m is processed to a tag-ciphertext pair with associated data ad, and a message m' is recovered from this pair using the same associated data ad, then the messages m, m' shall be identical. This is formalized via the FUNC game in Fig. 3.1. In particular, observe that if the adversary queries Dec(ad, c) (for the authentic ad and c that it receives in line 02) and the dec procedure produces output m', the game promises that $m' = m$ (lines 05,06). Recall from Sec. 1.3 that this means the game stops with output 1 if $m' \neq m$. Intuitively, the scheme is *functional* if we can rely on $m' = m$, that is, if the maximum advantage $\mathbf{Adv}^{\text{func}}(\mathcal{A}) := \max_{ad \in \mathcal{AD}, m \in \mathcal{M}} \mathbb{P}[\text{FUNC}(ad, m, \mathcal{A})]$ that can be attained by realistic adversaries $\mathcal{A}$ is negligible. The scheme is perfectly functional if $\mathbf{Adv}^{\text{func}}(\mathcal{A}) = 0$ for all $\mathcal{A}$. We remark that the universal quantification over all pairs (ad, m) makes our advantage definition particularly robust.

Our security notions demand that the integrity of ciphertexts be protected (INT-CTXT), and that encryptions be indistinguishable in the presence of chosen-ciphertext attacks (IND-CCA). The notions are formalized via the INT and $\text{IND}^0, \text{IND}^1$ games in Fig. 3.1, where the latter two depend on some equivalence relation $\equiv \subseteq \mathcal{M} \times \mathcal{M}$ on the message space.[18] For consistency, in lines 07,15,24 we suppress the message in all games if the adversary queries $\text{Dec}(ad, c)$. This is crucial in the IND^b games, as otherwise the adversary would trivially learn which message was encrypted, but does not harm in the other games as the adversary already knows m. Recall from Sec. 1.3 that all algorithms can abort, and if they do, then the oracles immediately abort. This property is crucial in the INT game where the dec algorithm must abort for unauthentic input such that the oracle immediately aborts. Otherwise, the game will reward the adversary, that is the game stops with 1 (line 14). We say that a scheme provides *integrity* if the maximum advantage $\mathbf{Adv}^{\text{int}}(\mathcal{A}) := \max_{ad \in \mathcal{AD}, m \in \mathcal{M}} \mathbb{P}[\text{INT}(ad, m, \mathcal{A})]$ that can be attained by realistic adversaries $\mathcal{A}$ is negligible, and that it provides *indistinguishability* if the same holds for the advantage $\mathbf{Adv}^{\text{ind}}(\mathcal{A}) := \max_{ad \in \mathcal{AD}, m^0, m^1 \in \mathcal{M}} |\mathbb{P}[\text{IND}^1(ad, m^0, m^1, \mathcal{A})] - \mathbb{P}[\text{IND}^0(ad, m^0, m^1, \mathcal{A})]|$.

3.3.2 Sufficiency of ETS for Outsourced Storage

We define the syntax of an outsourced storage scheme. We model such a scheme as a set of stateful algorithms, where algorithm write is invoked to store data and algorithm read is invoked to retrieve it. We indicate the statefulness of the algorithms by appending the term $\langle st \rangle$ to their names, where st is the state variable.

Definition 3.3.2. *Let $\mathcal{M}$ be a message space. A storage outsourcing scheme for $\mathcal{M}$ consists of algorithms* $\text{gen}, \text{write}, \text{read}$*, a state space $\mathcal{ST}$, and a ciphertext space $\mathcal{C}$. The state generation algorithm* gen *takes no input and outputs an (initial) state $st \in \mathcal{ST}$. The storage algorithm* write *takes a state $st \in \mathcal{ST}$ and a message $m \in \mathcal{M}$, and outputs an (updated) state $st \in \mathcal{ST}$ and a ciphertext $c \in \mathcal{C}$. The retrieval algorithm* read *takes a state $st \in \mathcal{ST}$ and a ciphertext $c \in \mathcal{C}$, and outputs an updated state $st \in \mathcal{ST}$ and a message $m \in \mathcal{M}$. A shortcut notation for this API is*

$$\text{gen} \to \mathcal{ST} \qquad \mathcal{M} \to \text{write}\langle \mathcal{ST} \rangle \to \mathcal{C} \qquad \mathcal{C} \to \text{read}\langle \mathcal{ST} \rangle \to \mathcal{M} \ .$$

[18] We use relation $\equiv$ (in line 18 of IND^b) to deal with certain restrictions that practical ETS schemes may feature. Concretely, our construction does not take effort to hide the length of encrypted messages, implying that indistinguishability is necessarily limited to same-length messages. In our formalization such a technical restriction can be expressed by defining $\equiv$ such that $m^0 \equiv m^1 \Leftrightarrow |m^0| = |m^1|$.

Correctness and Security. We require of a storage outsourcing scheme that if a message m is processed to a ciphertext, and subsequently a message m' is recovered from this ciphertext, then the messages m, m' shall be identical. This is formalized via the FUNC game in Fig. 3.2. Observe boolean flag *is* ('in-sync') tracks whether the attack is active or passive. Initially $is = \mathtt{T}$, i.e., the attack is passive; however, once the adversary requests the reading of a ciphertext that is not the last created one, the game sets $is \leftarrow \mathtt{F}$ to flag the attack as active (line 11). For passive attacks the game promises that any m returned by the read procedure is the last one that was processed by the write procedure (line 13). Intuitively, the scheme is *functional* if the maximum advantage $\mathbf{Adv}^{\text{func}}(\mathcal{A}) := \mathbb{P}[\text{FUNC}(\mathcal{A})]$ that can be attained by realistic adversaries $\mathcal{A}$ is negligible. The scheme is perfectly functional if $\mathbf{Adv}^{\text{func}}(\mathcal{A}) = 0$ for all $\mathcal{A}$.

Our security notions demand that the integrity of ciphertexts be protected (INT-CTXT), and that encryptions be indistinguishable in the presence of chosen-ciphertext attacks (IND-CCA). The notions are formalized via the INT and $\text{IND}^0, \text{IND}^1$ games in Fig. 3.2, where the latter two depend on some equivalence relation $\equiv \subseteq \mathcal{M} \times \mathcal{M}$ on the message space (see also Footnote 18). Recall from Sec. 1.3 that all algorithms can abort, and if they do, the oracles immediately abort. This property is crucial in the INT game where the read algorithm must abort for unauthentic input such that the adversary is not rewarded in the subsequent line in the Read oracle. For consistency we suppress the message in the Read oracle for passive attacks in all games if the adversary queries $\text{Dec}(ad, c)$. This is crucial in the IND^b games, as otherwise the adversary would trivially learn which message was encrypted, but does not harm in the other games as the adversary already knows m for passive attacks. Furthermore, we remark the adversary is only allowed to query the Corrupt oracle if M contains at most 1 message, i.e., the ChWrite oracle was queried for $m^0 = m^1$. Otherwise, the adversary would be able to run the read procedure and trivially learn m. We say that a scheme provides *integrity* if the maximum advantage $\mathbf{Adv}^{\text{int}}(\mathcal{A}) := \mathbb{P}[\text{INT}(\mathcal{A})]$ that can be attained by realistic adversaries $\mathcal{A}$ is negligible, and that it provides *indistinguishability* if the same holds for the advantage $\mathbf{Adv}^{\text{ind}}(\mathcal{A}) := |\mathbb{P}[\text{IND}^1(\mathcal{A})] - \mathbb{P}[\text{IND}^0(\mathcal{A})]|$.

Construction from ETS. Constructing secure outsourced storage from ETS is immediate: The write procedure samples a uniformly random key and runs the enc procedure of ETS to obtain a binding tag and ciphertext. It stores the binding tag (and key) in the state and returns the ciphertext. The read procedure gets the key and binding tag from the state, runs the dec procedure of ETS and returns the message. The details of this construction are in Fig. 3.3. The security argument is obvious.

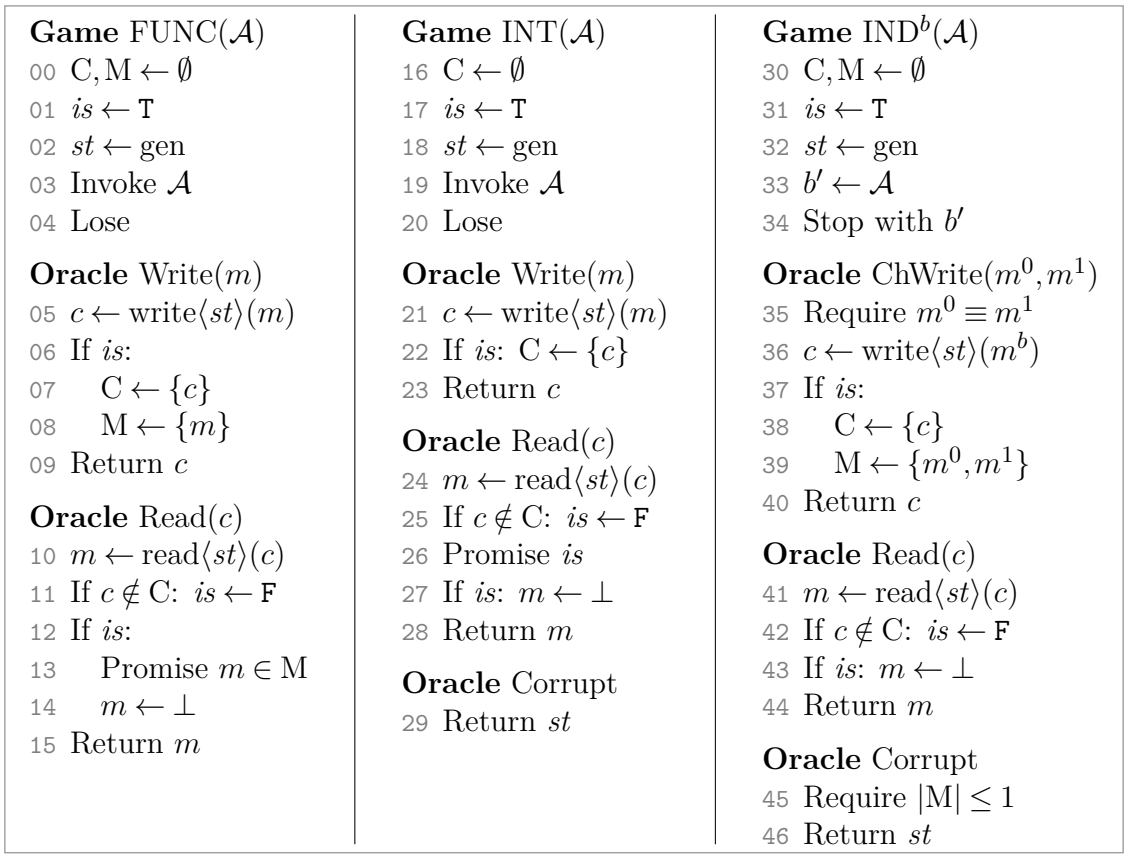

Fig. 3.2 Games for outsourced storage. For all values m, m^0, m^1, c provided by the adversary we require that $m, m^0, m^1 \in \mathcal{M}$ and $c \in \mathcal{C}$. Assuming $\perp \notin \mathcal{M}$, we encode suppressed messages with $\perp$. Boolean flag is ('in-sync') tracks whether the attack is active or passive. For the meaning of instructions Stop with, Lose, Promise, and Require see Sec. 1.3.

3.4 Construction of Encrypt-to-Self

We mentioned in Sec. 3.1 that a generic construction of ETS can be realized by combining standard symmetric encryption with a cryptographic hash function: one encrypts the message and computes the binding tag as the hash of the ciphertext. Here we provide a more efficient construction that builds on the compression function of a Merkle–Damgård hash function. To be more precise, our construction uses a tweakable compression function with tweak space $T = \{0,1\}$, i.e., the domain of the compression function is extended by one bit (see Sec. 3.2.2). We provide a general definition below.

Definition 3.4.1. *For Σ an alphabet, $c, d \in \mathbb{N}^+$ with $c \leq d$, and a tweak space T, we define a tweakable compression function to be a function $F\colon \Sigma^d \times T \times \Sigma^c \to \Sigma^c$ that takes*

Proc gen	**Proc** write$\langle st\rangle(m)$	**Proc** read$\langle st\rangle(c)$
00 $\mathrm{S} \leftarrow \emptyset$	03 $k \leftarrow_{\$} \mathcal{K}$	07 Require $\mathrm{S} \neq \emptyset$
01 $st := \mathrm{S}$	04 $(bt, c) \leftarrow \mathrm{enc}(k, \epsilon, m)$	08 $\{(k, bt)\} \leftarrow \mathrm{S}$
02 Return st	05 $\mathrm{S} \leftarrow \{(k, bt)\}$	09 $m \leftarrow \mathrm{dec}(k, bt, \epsilon, c)$
	06 Return c	10 Return m

Fig. 3.3 Construction for outsourced storage from ETS. If in line 07 the condition is not met, the read algorithm aborts with some error indicator. Recall from Sec. 1.3 that the read algorithm also aborts if the dec invocation in line 09 aborts.

as input a block $B \in \Sigma^d$ from the data domain, a tweak $t \in T$ from the tweak space, and a string $C \in \Sigma^c$ from the chain domain, and outputs a string $C' \in \Sigma^c$ in the chain domain.

We will write $F_t(B, C)$ as shorthand notation for $F(B, t, C)$. For practical tweakable compression functions the memory alignment value mav (see Sec. 3.2.1) will divide both c and d. When constructing an ETS scheme from F, because the compression function only takes fixed-size input, we need to map the (ad, m) input to a series of block–tweak pairs (B, t). We will refer to this mapping as the input *encoding*. We take a modular approach by fixing the encoding independently of the encryption engine, and detail the former in Sec. 3.4.1 and the latter in Sec. 3.4.2. Together they form an efficient construction of ETS.

We first convey a rough overview of our ETS construction. In Fig. 3.4 we consider an example with block size d double the chaining value size c. We assume that key k is padded to size d. The first block B_1 only contains associated data and we XOR B_1 with the key k before we feed it into the compression function. From the second block, we start processing message data. We fill the first half of the block with associated data ad_3 and the second half with message data m_1, and XOR with the key. We observe that in most uses cases of EtS it should be expected that the associated data string is shorter than the message input. Because we will pad with null bytes, in terms of Fig. 3.4 this means that the ad-input of most compression function invocations will be constant and no XORing is necessary once the associated data is fully processed as the first component of the concatenation remains invariant and can be precomputed. We XOR m_1 with the current chaining value C_1, to generate a partial ciphertext ct_1. The same happens in the third block and we append ct_2 to the ciphertext. If there is associated data left after processing all message data we can load the entire block with associated data, which occurs in the fourth block. Note, we no longer need to XOR the key into the block after we have processed all message data, because at this point the input to the compression function will already be independent of the message m. After processing all blocks, we XOR an offset $\omega \in \{\omega_0, \omega_1\}$ with the chaining value, where ω_0, ω_1 are two distinct

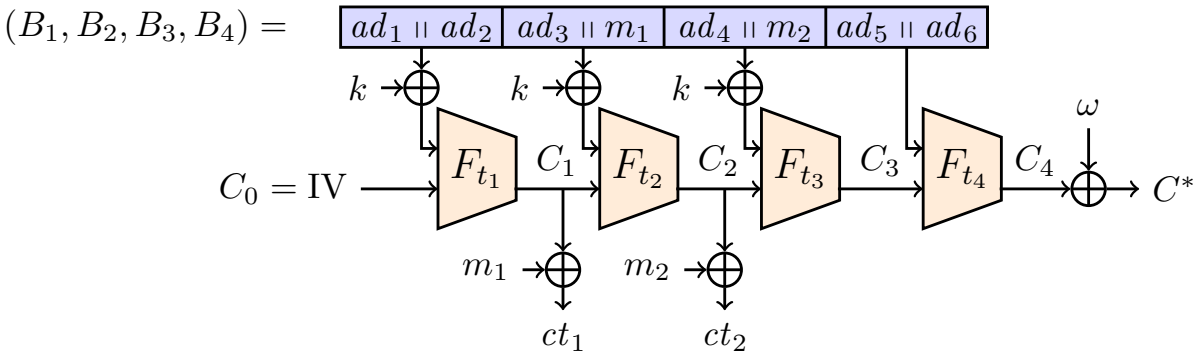

Fig. 3.4 Example for $\mathrm{enc}(k, ad, m)$ where $d = 2c$ and $ad = ad_1 \mid\mid \ldots \mid\mid ad_6$ and $m = m_1 \mid\mid m_2$ with $|ad| = 6c$ and $|m| = 2c$. For clarity we have made the blocks B_i, as they are output by the encoding function, explicit. Inspiration for this figure is drawn from https://www.iacr.org/authors/tikz/.

constants. The binding tag will be (a truncation of) the last chaining value C^*.[19] Note that the task of the encoding is not only to partition ad and m into blocks $B_1, B_2, \ldots$ as described, but also to derive tweak values $t_1, t_2, \ldots$ and the choice of the final offset ω in such a way that the overall encoding is injective.

3.4.1 Message Block Encoding

We turn to the technical component of our ETS construction that encodes the (ad, m) input into a series of output pairs (B, t) and the final offset value ω. For authenticity we require that the encoding is injective. For efficiency we require that the encoding is online (i.e., the input is read only once, left-to-right, and with small state), that the number of output pairs is as small as possible, and that the encoding preserves memory alignment (see Sec. 3.2.1). Syntactically, for the outputs we require that all $B \in \Sigma^d$, all $t \in T$, and $\omega \in \Omega$, where quantities c, d are those of the employed compression function, $T = \{0,1\}$, and $\Omega \subseteq \Sigma^c$ is any two-element set. (Note that $|T| = 2$ allows us to use the tweaking approach from Sec. 3.2.2; further, in our implementations we use $\Omega = \{\omega_0, \omega_1\}$ where $\omega_0 = \texttt{0x00}^c$ and $\omega_1 = \texttt{0xa5}^c$.) Overall, the task we are facing is the following:

Task. Assume $|\Sigma| = 256$ and $\mathcal{AD} = \mathcal{M} = \Sigma^*$ and $T = \{0,1\}$ and $|\Omega| = 2$. For $c, d \in \mathbb{N}^+$, $c < d$, find an injective encoding function $\mathrm{encode}\colon \mathcal{AD} \times \mathcal{M} \to (\Sigma^d \times T)^* \times \Omega$ that takes as input two finite strings and outputs a finite sequence of pairs $(B, t) \in \Sigma^d \times T$ and an offset $\omega \in \Omega$.

[19] It will be crucial to fix ω_0, ω_1 such that they are distinct also after truncation.

A detailed specification of our encoding (and decoding) function can be found in Fig. 3.6, but we present it here in text. Our construction does not use the decoding function, but we provide it anyway to show that the encoding function is indeed injective. Roughly, we encode as follows. We fill the first block with associated data and for any subsequent block we load the associated data in the first part of the block and the message in the second part of the block. When we have processed all the message data, we load the full block with ad again. Clearly, we need to pad ad if it runs out before we have processed all message data. We do this by appending a special termination symbol $\diamond \in \Sigma$ to ad and then appending null bytes as needed. Similarly, we need to pad the message if the message length is not a multiple of c. Naturally, one might want to pad the message to a multiple of c. However, this is suboptimal: Consider the scenario where there are $d-c+1$ bytes remaining to be processed of associated data and 1 byte of message data. In principle, message and associated data would fit into a single block, but this would not be the case any longer if the message is padded to size c. On the other hand, for efficiency reasons we do not want to misalign all our remaining associated data. If we do not pad at all, when we process the next d bytes of associated data, we can only fit $d-1$ bytes in the block and have to put 1 byte into the next block. Therefore, we pad m up to a multiple of the memory alignment value mav. To be precise, we pad message with null bytes until reaching a multiple of mav. We replace the final (null) byte with the message length $|m|$; this will uniquely determine where m stops and the padding begins. This restricts us to $c \leq 256$ bytes such that $|m|$ always can be encoded into a single byte. As far as we are aware, any current practical compression function satisfies this requirement.

In Fig. 3.5, for the artificially small case with $c = 1$ and $d = 2$ we provide four examples of what the blocks would look like for different inputs. The top row shows the encoding of 8 bytes of associated data and an empty message. The second row shows the encoding of empty associated data and 3 bytes of message data. The third row shows the encoding of 6 bytes of associated data and 2 bytes of message data. The final row shows the encoding of 3 bytes of associated data and 3 bytes of message data.

We have two ambiguities remaining. (1) How to tell whether ad was padded or not? Consider the first row in Fig. 3.5. What distinguishes the case $ad = ad_1 \mid\mid \ldots \mid\mid ad_7$ from $ad = ad_1 \mid\mid \ldots \mid\mid ad_7 \mid\mid ad_8$ with $ad_8 = \diamond$? A similar question applies to the message. (2) How to tell whether a block contains message data or not? Compare e.g., the first row with the third row. This is where the tweaks come into play.

First of all, we tweak the first block if and only if the message is empty. This fully separates the authentication-only case from the case where we have message input.

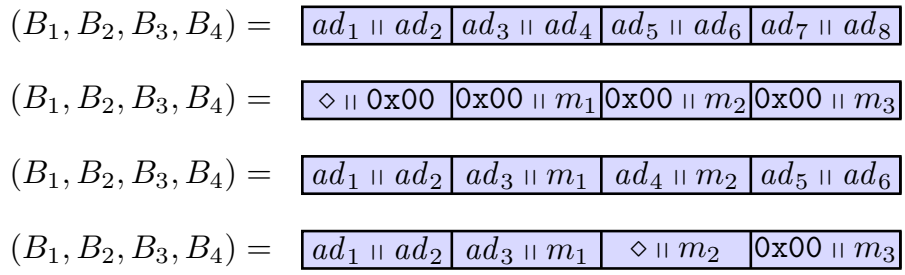

Fig. 3.5 Example encodings for the case $c = 1$ and $d = 2$.

Next, if the message is non-empty, we use the tweaks to indicate when we switch from processing message data to ad-only: we tweak when we have consumed all of m, but still have ad left. Note the first block never processes message data, so the earliest block this may tweak is the second block and hence this rule does not interfere with the first rule. Furthermore, observe this rule never tweaks the final block, as by definition of being the final block, we do not have any associated data left to process.

Next, we need to distinguish between the cases whether m is padded or not. In fact, as the empty message was already taken care of, we need to do this only if m is at least one byte in size. As in this case the final block does not coincide with the first block, we can exploit that its tweak is still unused; we correspondingly tweak the final block if and only if m is padded. Obviously, this does not interfere with the previous rules.

Finally, we need to decide whether ad was padded or not. We do not want to enforce a policy of 'always pad', as this could result in an extra block and hence an extra compression function invocation. Instead, we use our offset output. We set the offset ω to ω_1 if ad was padded; otherwise we set it to ω_0.

This completes our description of the encoding function. The decoding function is a technical exercise carefully unwinding the steps taken in the encoding function, which we perform in Fig. 3.6. We obtain that for all $m \in \mathcal{M}, ad \in \mathcal{AD}$ we have $\text{decode}(\text{encode}(ad, m)) = (ad, m)$. It immediately follows that our encoding function is injective. For readability we have implemented the core functionality of the encoding in a coroutine called nxt, rather than a subroutine. Instead of generating the entire sequence of (B, t) pairs and returning the result, it will 'Yield' one pair and suspend its execution. The next time it is called (e.g., the next step in a for loop), it will resume execution from where it called 'Yield', instead of at the beginning of the function, with all of its state intact. The encode procedure is a simple wrapper that runs the nxt procedure and collects its output, but our authenticated encryption engine described in Sec. 3.4.2 will call the nxt procedure directly.

Proc encode(ad, m)
00 $\mathrm{S}[\cdot] \leftarrow \cdot$; $i \leftarrow 0$
01 For $(B,t) \in \mathrm{nxt}(ad,m)$:
02 If $B \neq \epsilon$:
03 $i \leftarrow i+1$
04 $\mathrm{S}[i] \leftarrow (B,t)$
05 Else: $\omega \leftarrow t$
06 Return (S,ω)

Proc decode(S,ω)
07 $ad \leftarrow \epsilon$; $m \leftarrow \epsilon$
08 $n \leftarrow |\mathrm{S}|$; $j \leftarrow |\mathrm{S}|$
09 If $n = 0$:
10 Return (ad, m)
11 For $i \leftarrow 1$ to n:
12 $(B_i, t_i) \leftarrow \mathrm{S}[i]$
13 For $i \leftarrow 1$ to $n-1$:
14 If $t_i = 1$: $j \leftarrow i$
15 $ad \xleftarrow{\shortparallel} B_1$
16 For $i \leftarrow 2$ to $j - t_n$:
17 $B_i \shortparallel B_i' \leftarrow_c B_i$
18 $ad \xleftarrow{\shortparallel} B_i$
19 $m \xleftarrow{\shortparallel} B_i'$
20 If $n > 1 \wedge t_n = 1$:
21 $B_j \shortparallel l \leftarrow_{d-1} B_j$
22 $p \leftarrow -l \bmod \mathrm{mav}$
23 $a \leftarrow d - l - p$
24 $ad \shortparallel B_j \leftarrow_a B_j$
25 $m \leftarrow_j B_j$
26 For $i \leftarrow j+1$ to n:
27 $ad \xleftarrow{\shortparallel} B_i$
28 If $\omega = \omega_1$:
29 Split $ad \shortparallel \diamond \shortparallel 0^* \leftarrow ad$
30 Return (ad, m)

Proc nxt(ad, m)
31 ad_padded $\leftarrow$ F
32 m_padded $\leftarrow [m = \epsilon]$
33 m_final $\leftarrow [m = \epsilon]$
34 $\omega \leftarrow \omega_0$; $n \leftarrow 0$
35 While $ad \neq \epsilon \vee m \neq \epsilon$:
36 $n \leftarrow n+1$
37 $(B_n, t_n) \leftarrow (\epsilon, 0)$
38 If $n = 1$:
39 $d' \leftarrow d$
40 else:
41 $j \leftarrow -|m| \bmod \mathrm{mav}$
42 $d' \leftarrow d - |m| - j$
43 If $|ad| < d'$:
44 If not ad_padded:
45 $\omega \leftarrow \omega_1$
46 $ad \xleftarrow{\shortparallel} \diamond$
47 ad_padded $\leftarrow$ T
48 $j \leftarrow d' - |ad|$
49 $ad \xleftarrow{\shortparallel} 0^j$
50 $B_n \shortparallel ad \leftarrow_{d'} ad$
51 If $n > 1 \wedge m \neq \epsilon$:
52 If $|m| < c$:
53 m_padded $\leftarrow 1$
54 $j \leftarrow -|m| \bmod \mathrm{mav}$
55 $m \leftarrow m \shortparallel 0^{j-1} \shortparallel |m|$
56 $l \leftarrow \min(c, |m|)$
57 $m' \shortparallel m \leftarrow_l m$
58 $B_n \xleftarrow{\shortparallel} m'$
59 If $m = \epsilon$:
60 If $ad = \epsilon$:
61 $t_n \leftarrow$ m_padded
62 Else:
63 $t_n \leftarrow$ m_final
64 m_final $\leftarrow 0$
65 Yield (B_n, t_n)
66 Yield (ϵ, ω)

Proc enc(k, ad, m)
67 $ct \leftarrow \epsilon$; $C \leftarrow \mathrm{IV}$; $i \leftarrow 0$
68 For $(B,t) \in \mathrm{nxt}(ad,m)$:
69 If $B \neq \epsilon$:
70 $i \leftarrow i+1$
71 If $i = 1 \vee m \neq \epsilon$:
72 $B \leftarrow B \oplus k$
73 If $i > 1 \wedge m \neq \epsilon$:
74 $j \leftarrow \min(c, |m|)$
75 $m' \shortparallel m \leftarrow_j m$
76 $C' \leftarrow_j C$
77 $ct \xleftarrow{\shortparallel} m' \oplus C'$
78 $C \leftarrow F_t(B, C)$
79 $bt \leftarrow_{\mathrm{taglen}} C \oplus t$
80 Return (bt, ct)

Proc dec(k, bt, ad, ct)
81 $m \leftarrow \epsilon$; $C \leftarrow \mathrm{IV}$; $i \leftarrow 0$
82 For $(B,t) \in \mathrm{nxt}(ad, ct)$:
83 If $B \neq \epsilon$:
84 $i \leftarrow i+1$
85 If $i = 1 \vee ct \neq \epsilon$:
86 $B \leftarrow B \oplus k$
87 If $i > 1 \wedge ct \neq \epsilon$:
88 If $|ct| \geq c$:
89 $ct' \shortparallel ct \leftarrow_c ct$
90 $m \xleftarrow{\shortparallel} ct' \oplus C$
91 $B \xleftarrow{\oplus} 0^{d-c} \shortparallel C$
92 Else:
93 $C' \leftarrow_{|ct|} C$
94 $m \xleftarrow{\shortparallel} ct \oplus C'$
95 $j \leftarrow -|m| \bmod \mathrm{mav}$
96 $a \leftarrow d - |m| - j$
97 $B \xleftarrow{\oplus} 0^a \shortparallel C' \shortparallel 0^j$
98 $C \leftarrow F_t(B, C)$
99 $bt' \leftarrow_{\mathrm{taglen}} C \oplus t$
100 If $bt' \neq bt$: Abort
101 Return m

Fig. 3.6 ETS construction: encoder, decoder, encryptor, and decryptor. (Procedure nxt is a coroutine for encode, enc, and dec, see text.) Using global constants mav, c, d, taglen, and IV.

3.4.2 Encryption Engine

We now turn our focus to the encryption engine. We assume that the associated data and message are present in encoded format, i.e., as a sequence of pairs (B,t), where $B \in \Sigma^d$ is a block and $t \in \{0,1\}$ is a tweak, and additionally an offset $\omega \in \{\omega_0,\omega_1\}$. To be precise, we will use the nxt procedure that generates the next (B,t) pair on the fly.

We specify the encryption and decryption algorithms in Fig. 3.6 and assume they are provided with a key of length d. As illustrated in Fig. 3.4, the main idea is to XOR the key with all blocks that are involved with message processing. For the skeleton of the construction, we initialize the chaining value C to IV and loop through the sequence of pairs (B,t) output by the encoding function, each iteration updating the chaining value $C \leftarrow F_t(B,C)$. We now describe each iteration of the enc procedure in more detail, where numbers in brackets refer to line numbers in Fig. 3.6. If the block is empty [69], we are in the final iteration and do not do anything. Otherwise, we check if we are in the first iteration or if we have message data left [71]. In this case we XOR the key into the block [72]. This ensures we start with an unknown input block and that subsequent inputs are statistically independent of the message block. If we only have ad remaining we can use the block directly as input to the compression function. If we have message data left we will encrypt it starting from the second block [73]. To encrypt, we take a chunk of the message, XOR it with the chaining value of equal size and append the result to the ciphertext [74–77]. We only start encrypting from the second iteration as the first chaining value is public. Finally, we call the compression function F_t to update our chaining value [78]. Once we have finished the loop, the last pair (B,t) equals (ϵ,ω) by definition. So we XOR the offset ω with the chaining value C and truncate the result to obtain the binding tag [79]. We return the binding tag along with the ciphertext.

The dec procedure is similar to the enc procedure but needs to be slightly adapted. Informally, the nxt procedure now outputs a block $B = (ad \mid\mid ct)$ [82] instead of $B = (ad \mid\mid m)$ [68]. Hence, we XOR with the chaining variable [91,97] such that the block becomes $B = (ad \mid\mid m)$ and the compression function call takes equal input compared to the enc procedure. The case distinction handles the slightly different positioning of ciphertext in the blocks. Finally, there obviously is a check if the computed binding tag is equal to the stored binding tag [100].

3.5 Security Proofs

In order to prove security, we need further assumptions on our compression function than the standard assumption of preimage resistance and collision resistance. For example,

we need F to be difference unpredictable. Roughly, this notion says it is hard to find a pair (x, y) such that $F(x) = F(y) \oplus z$ for a given difference z. Moreover, we truncate the binding tag, so actually it should be hard to find a tuple such that this equation holds for the first $|bt|$ bits. We note collision resistance of F does not imply collision resistance of a truncated version of F [21]. However, such assumptions could be justified when one considers the compression function as a random function. Hence, instead of several ad hoc assumptions, we prove our construction secure directly in the random oracle model.

As described in Sec. 3.2.2 we tweak the SHA2 compression function by modifying the chaining value depending on the tweak. Let F be the tweakable compression function in Fig. 3.6, We write F' for the SHA2 compression function that will take as input the block and the (modified) chaining value. Let $H\colon \Sigma^d \times \Sigma^c \to \Sigma^c$ be a random oracle. In the security analysis of the SHA2 construction, we will substitute H for F' in our construction.

We remark the BLAKE2b compression function is a tweakable compression function and it can be substituted directly for a random oracle with an extended input space. That is, a random oracle $\bar{H}\colon \Sigma^d \times \{0,1\} \times \Sigma^c \to \Sigma^c$. Hence, in the security analysis of the BLAKE2b construction, we will substitute $\bar{H}$ for F in our construction.

We remark that we cannot treat our tweaked SHA2 compression function F in this way as it would be distinguishable from random oracle $\bar{H}$. To see this, observe that querying F on the unmodified chaining variable with tweak $t = 1$ yields the same result as querying F on the modified chaining variable with $t = 0$. In the random oracle $\bar{H}$ these two queries are completely independent.

In the random oracle model, our ETS construction from Fig. 3.6 with a non-tweakable / tweakable compression function provides integrity (Thm 3.5.1 / Thm 3.5.3) and indistinguishability (Thm 3.5.2 / Thm 3.5.4), assuming sufficiently large tag and key lengths. We detail the full theorem statements and security proofs below. We will first discuss the non-tweakable compression function instantiation and subsequently the tweakable compression function instantiation.

3.5.1 Non-Tweakable Compression Function

Let $H\colon \Sigma^d \times \Sigma^c \to \Sigma^c$ be a random oracle. Recall we consider an instantiation with a standard (non-tweakable) compression function F' transformed as described in Sec. 3.2.2 into a tweakable compression function F. We replace F', used internally by F, with random oracle H.

Theorem 3.5.1. *Let π be the construction given in Fig. 3.6, H a random oracle replacing the compression function, $\mathcal{A}$ an adversary, $\mathbf{Adv}^{\mathrm{int}}_{\pi}(\mathcal{A})$ the advantage that $\mathcal{A}$ has against π in the integrity game of Fig. 3.1 and q the number of random oracle queries (either directly or indirectly via* Dec*). We have,*

$$\mathbf{Adv}^{\mathrm{int}}_{\pi}(\mathcal{A}) \leq q^2 \cdot 2^{-c} + q \cdot 2^{-|bt|}.$$

Proof. For all $ad \in \mathcal{AD}, m \in \mathcal{M}$ we will show that

$$\mathbb{P}[\mathrm{INT}(ad, m, \mathcal{A})] \leq q^2 \cdot 2^{-c} + q \cdot 2^{-|bt|}.$$

Let $ad \in \mathcal{AD}$ be associated data and let $m \in \mathcal{M}$ be a message. The game $\mathrm{INT}(ad, m, \mathcal{A})$ samples a uniformly random key $k \in \mathcal{K}$ and computes $(bt, c) = \mathrm{enc}(k, ad, m)$. $\mathcal{A}$ wins the INT game if it provides a pair $(ad', c') \neq (ad, c)$ such that $\mathrm{dec}(k, bt, ad', c')$ succeeds, which only happens if $bt' = bt$. Recall the encoding function outputs a sequence S of (B, t) pairs and an offset ω. Because the encoding function is injective we must have $S' \neq S$ or $\omega' \neq \omega$. Let us first assume $S' = S$. Let C_n denote the final chaining variable. Because the sequences are equal, we will arrive at $C'_n = C_n$. We must have $\omega' \neq \omega$, but clearly $C_n \oplus \omega_0$ is not equal to $C_n \oplus \omega_1$ (even after truncation), that is, $bt' \neq bt$. We have a contradiction and conclude $S' \neq S$.

For the case $S' \neq S$, let us now assume the subcase $\omega' \neq \omega$. The first $|bt|$ bits of $C'_{n'}$ must equal the first $|bt|$ bits of $C_n \oplus \omega \oplus \omega'$, i.e., $\mathcal{A}$ must find a partial preimage. Because H is a random oracle, $\mathcal{A}$ would succeed with probability at most $q \cdot 2^{-|bt|}$, where q is the number of queries. In the other subcase we have $\omega' = \omega$. Then the first $|bt|$ bits of $C'_{n'}$ must equal the first $|bt|$ bits of C_n, i.e., the first $|bt|$ bits of $H(B'_{n'}, \hat{C}'_{n'-1})$ must equal the first $|bt|$ bits of $H(B_n, \hat{C}_{n-1})$, where $\hat{C}'_{n'-1}, \hat{C}_{n-1}$ are the chaining values $C'_{n'-1}, C_{n-1}$ after applying tweaks $t'_{n'}, t_n$, respectively. If the inputs are not equal, $\mathcal{A}$ has found a partial second preimage. Since H is a random oracle, $\mathcal{A}$ would succeed with probability at most $q \cdot 2^{-|bt|}$, where q is the number of oracle queries. However, if the inputs are equal we know $\hat{C}'_{n'-1} = \hat{C}_{n-1}$. Let us write $\hat{C}'_{n'-1} = C'_{n'-1} \oplus \tau'$ and $\hat{C}_{n-1} = C_{n-1} \oplus \tau$. We obtain $C'_{n'-1} = C_{n-1} \oplus \tau \oplus \tau'$. Thus, either $\mathcal{A}$ has found a collision or $C'_{n'-1} = C_{n-1}$. We can repeat the argument to reason about C'_{n-2}, C_{n-2}, etc. By a standard birthday argument we can bound the probability of a collision by $q^2 \cdot 2^{-c}$.

If we eventually conclude $C'_{n'-\delta} = C_{n-\delta} = \mathrm{IV}$, we know one of the sequences is longer, i.e., $n' - \delta > 0$ or $n - \delta > 0$. Otherwise the sequences would be equal, which is excluded by the injectivity of the encoding function. In the case $n - \delta > 0$, there has been a collision in the hash function, we have already bounded this probability above. Thus, let us

assume $n' - \delta > 0$. We have $H(B_{n'-\delta}, \hat{C}_{n'-\delta-1}) = \text{IV}$. Thus $\mathcal{A}$ has found a preimage of IV. Because H is a random oracle, $\mathcal{A}$ would succeed with probability at most $q \cdot 2^{-c}$. □

Theorem 3.5.2. *Let π be the construction given in Fig. 3.6, H a random oracle replacing the compression function, $\mathcal{A}$ an adversary, $\mathbf{Adv}^{\text{ind}}_{\pi}(\mathcal{A})$ the advantage that $\mathcal{A}$ has against π in the indistinguishability games of Fig. 3.1 and q the number of random oracle queries (either directly or indirectly via* Dec*). We have,*

$$\mathbf{Adv}^{\text{ind}}_{\pi}(\mathcal{A}) \leq q^2 \cdot 2^{-c} + q \cdot 2^{-|k|} + \mathbf{Adv}^{\text{int}}_{\pi}(\mathcal{A}).$$

Proof. Other than the challenge pair (ad, c), we can assume the decryption oracle rejects all queries by $\mathcal{A}$. Otherwise $\mathcal{A}$ would immediately win the integrity game and the theorem holds. Encryption is done by XORing the message with the chaining variable. As long as the chaining variable never repeats, each input to H is a fresh query that has not been seen before. Then H will provide fresh, uniformly random output, as it is a random oracle. By a standard birthday argument we can bound the probability of a collision by $q^2 \cdot 2^{-c}$. Now let us assume there is no collision. Each chaining variable that is used to encrypt is output of a query to H that XORed the key k with the input. Additionally each block that has message data as input is also XORed with the key k. Thus if $\mathcal{A}$ does not know k it cannot query H to obtain the chaining variable. The key is only used with input to the compression function, and since H is a random oracle, $\mathcal{A}$ can only learn by guessing the input and checking the random oracle output. However, this has a success probability of at most $q \cdot 2^{-|k|}$. □

3.5.2 Tweakable Compression Function

Let $\bar{H} \colon \Sigma^d \times \{0,1\} \times \Sigma^c \to \Sigma^c$ be a random oracle. We now consider an instantiation with a tweakable compression function F. We replace F with random oracle $\bar{H}$.

Theorem 3.5.3. *Let π be the construction given in Fig. 3.6, $\bar{H}$ a random oracle replacing the tweakable compression function, $\mathcal{A}$ an adversary, $\mathbf{Adv}^{\text{int}}_{\pi}(\mathcal{A})$ the advantage that $\mathcal{A}$ has against π in the integrity game of Fig. 3.1 and q the number of random oracle queries (either directly or indirectly via* Dec*). We have,*

$$\mathbf{Adv}^{\text{int}}_{\pi}(\mathcal{A}) \leq q^2 \cdot 2^{-c} + q \cdot 2^{-|bt|}.$$

Proof. For all $ad \in \mathcal{AD}, m \in \mathcal{M}$ we will show that

$$\mathbb{P}[\text{INT}(ad, m, \mathcal{A})] \leq q^2 \cdot 2^{-c} + q \cdot 2^{-|bt|}.$$

Let $ad \in \mathcal{AD}$ be associated data and let $m \in \mathcal{M}$ be a message. The game $\mathrm{INT}(ad, m, \mathcal{A})$ samples a uniformly random key $k \in \mathcal{K}$ and computes $(bt, c) = \mathrm{enc}(k, ad, m)$. $\mathcal{A}$ wins the INT game if it provides a pair $(ad', c') \neq (ad, c)$ such that $\mathrm{dec}(k, bt, ad', c')$ succeeds, which only happens if $bt' = bt$. Recall the encoding function outputs a sequence S of (B, t) pairs and an offset ω. Because the encoding function is injective we must have $S' \neq S$ or $\omega' \neq \omega$. Let us first assume $S' = S$. Let C_n denote the final chaining variable. Because the sequences are equal, we will arrive at $C'_n = C_n$. We must have $\omega' \neq \omega$, but clearly $C_n \oplus \omega_0$ is not equal to $C_n \oplus \omega_1$ (even after truncation), that is, $bt' \neq bt$. We have a contradiction and conclude $S' \neq S$.

For the case $S' \neq S$, let us now assume the subcase $\omega' \neq \omega$. The first $|bt|$ bits of $C'_{n'}$ must equal the first $|bt|$ bits of $C_n \oplus \omega \oplus \omega'$, i.e., $\mathcal{A}$ must find a partial preimage. Because $\bar{H}$ is a random oracle, $\mathcal{A}$ would succeed with probability at most $q \cdot 2^{-|bt|}$, where q is the number of queries. In the other subcase we have $\omega' = \omega$. Then the first $|bt|$ bits of $C'_{n'}$ must equal the first $|bt|$ bits of C_n, i.e., the first $|bt|$ bits of $\bar{H}(B'_{n'}, t_{n'}, C'_{n'-1})$ must equal the first $|bt|$ bits of $\bar{H}(B_n, t_n, C_{n-1})$. If the inputs are not equal, $\mathcal{A}$ has found a partial second preimage. Since $\bar{H}$ is a random oracle, $\mathcal{A}$ would succeed with probability at most $q \cdot 2^{-|bt|}$, where q is the number of oracle queries. However, if the inputs are equal we know $C'_{n'-1} = C_{n-1}$. Thus, either $\mathcal{A}$ has found a collision or $C'_{n'-1} = C_{n-1}$. We can repeat the argument to reason about C'_{n-2}, C_{n-2}, etc. By a standard birthday argument we can bound the probability of a collision by $q^2 \cdot 2^{-c}$.

If we eventually conclude $C'_{n'-\delta} = C_{n-\delta} = \mathrm{IV}$, we know one of the sequences is longer, i.e., $n' - \delta > 0$ or $n - \delta > 0$. Otherwise the sequences would be equal, which is excluded by the injectivity of the encoding function. In the case $n - \delta > 0$, there has been a collision in the hash function, we have already bounded this probability above. Thus, let us assume $n' - \delta > 0$. We have $\bar{H}(B_{n'-\delta}, t_{n'-\delta}, C_{n'-\delta-1}) = \mathrm{IV}$. Thus $\mathcal{A}$ has found a preimage of IV. Because $\bar{H}$ is a random oracle, $\mathcal{A}$ would succeed with probability at most $q \cdot 2^{-c}$. □

Theorem 3.5.4. *Let π be the construction given in Fig. 3.6, $\bar{H}$ a random oracle replacing the tweakable compression function, $\mathcal{A}$ an adversary, $\mathbf{Adv}^{\mathrm{ind}}_{\pi}(\mathcal{A})$ the advantage that $\mathcal{A}$ has against π in the indistinguishability games of Fig. 3.1 and q the number of random oracle queries (either directly or indirectly via* Dec*). We have,*

$$\mathbf{Adv}^{\mathrm{ind}}_{\pi}(\mathcal{A}) \leq q^2 \cdot 2^{-c} + q \cdot 2^{-|k|} + \mathbf{Adv}^{\mathrm{int}}_{\pi}(\mathcal{A}).$$

Proof. Other than the challenge pair (ad, c), we can assume the decryption oracle rejects all queries by $\mathcal{A}$. Otherwise $\mathcal{A}$ would immediately win the integrity game and the proposition holds. Encryption is done by XORing the message with the chaining variable.

As long as the chaining variable never repeats, each input to $\bar{H}$ is a fresh query that has not been seen before. Then $\bar{H}$ will provide fresh, uniformly random output, as it is a random oracle. By a standard birthday argument we can bound the probability of a collision by $q^2 \cdot 2^{-c}$. Now let us assume there is no collision. Each chaining variable that is used to encrypt is output of a query to $\bar{H}$ that XORed the key k with the input. Additionally each block that has message data as input is also XORed with the key k. Thus if $\mathcal{A}$ does not know k it cannot query $\bar{H}$ to obtain the chaining variable. The key is only used with input to the compression function, and since $\bar{H}$ is a random oracle, $\mathcal{A}$ can only learn by guessing the input and checking the random oracle output. However, this has a success probability of at most $q \cdot 2^{-|k|}$. □

3.6 Implementation of Encrypt-to-Self

We implemented three versions of the EtS primitive. We developed optimized C code for the padding scheme and encryption engine from Fig. 3.6, based on the compression functions of common hash functions. Specifically, our EtS implementations are based on the compression functions of SHA256, SHA512, and BLAKE2 [80, 89]. We chose these functions as all three of them are ARX designs (Add–Rotate–Xor) which makes them particularly efficient in software implementations. While SHA256 and SHA512 are more widely standardized and used than BLAKE2, only the latter is a HAIFA construction and tweakable without ad-hoc modifications. Note that due to the used internal register size of 32 bits, SHA256 is most competitive on 32-bit CPUs; in contrast, SHA512 and BLAKE2 use 64-bit registers and thus perform best on 64-bit CPUs.

We implemented all components of EtS in plain C, including the compression functions, the encoding schemes, and the EtS framework. In addition we implemented a range of self-tests and provide test vectors. We note that while in particular the compression functions would be good candidates for being re-implemented in assembly for further efficiency improvements, we believe that, as all three compression functions are ARX designs, the penalty of not hand-optimizing is not too drastic.

We released the source code of our implementation as open source software. The terms of use are those granted by the Apache license[20]. The code is available at https://github.com/cryptobertram/encrypt-to-self.

We conducted timing measurements for our implementations. We measured on two devices: on a roughly ten year old CPU (Q4'11) that identifies itself as `Intel Core i3-2350M CPU @ 2.30GHz`, and on a more recent CPU (Q1'17) of the type `Intel Core`

[20] https://www.apache.org/licenses/LICENSE-2.0

`i5-7300U CPU @ 2.60GHz`. The results are shown in Table 3.1. The timings were taken for various message lengths, with a 16 byte associated data input in all cases. Note that the BLAKE2 based version clearly outperforms the others for all tested message lengths. Further, SHA512 is generally faster than SHA256 (except for messages that are so short that one SHA256 compression function invocation is sufficient to fully encrypt the message). On the `i5-7300U` we compare these results versus the naive approach of encrypt-then-hash. For the naive approach we directly call the OpenSSL API to encrypt (AES256 in CBC mode) the message and to hash (SHA256) the ciphertext, ignoring the associated data input for simplicity. We observe that our BLAKE2 based version remains the fastest by a significant margin. While we concede that for large messages the naive approach becomes faster than our SHA256 implementation, we highlight that we are comparing our prototype versus a well established library.

Table 3.1 Timings (in microseconds) of EtS implementation

compression function	message length	time on i3-2350M	time on i5-7300U
SHA256	16	1.578	0.684
SHA512	16	2.101	0.881
BLAKE2	16	0.766	0.366
Naive	16		1.337
SHA256	48	2.354	1.014
SHA512	48	2.186	0.882
BLAKE2	48	0.767	0.372
Naive	48		1.549
SHA256	256	6.894	2.987
SHA512	256	5.054	2.139
BLAKE2	256	1.805	0.858
Naive	256		2.197
SHA256	1024	25.590	10.860
SHA512	1024	17.380	7.213
BLAKE2	1024	6.040	2.846
Naive	1024		4.714

Chapter 4

PERKS: Persistent and Distributed Key Acquisition for Secure Storage from Passwords

Passwords are ubiquitous as authentication tokens and yet constructing schemes based on passwords is notoriously difficult to get right. Users regularly re-use and/or forget their passwords, application servers store passwords incorrectly, and more and more physical and technical tools are needed to prevent attacks and misuse. We consider the general problem of converting a human-memorable password into a single cryptographic secret, with minimal storage and communication requirements.

4.1 Introduction

As a motivating case study, consider instant messaging (IM) applications: it is at present not clear what keying material should be used to encrypt messages and files that are stored on the user's device, and/or backed up to an external backup service or cloud storage provider (CSP)[1]. It is not clear what security properties are obtainable in the scenario where a user defends against potential loss of their device by backing up messages, never mind in each of the many other possible variations of the message storage scenario (long-term on-device encryption, temporary backup for 'device changeover', backup for immediate local deletion). Our solutions are targeted at this main scenario—where a user acquires a new phone and wants to recover their backed-up data using only their

[1] In commercial settings there may exist on-premise file/backup storage, but in our more general case the entity storing the ciphertexts is regarded as external.

password—but with applications to the others[2]. In this setting a user may interact with their IM service, a CSP that stores messages and media, and potentially other services that contribute to keying material: the user would prefer not to (fully) trust all of these services (or their device) and additionally would like to draw entropy from each of these services in deriving a(n initial) key for data encryption. We refer to all of these n entities, potentially including the IM and outsourced storage providers, as *key-contributing servers*. Our key tool is an oblivious PRF (OPRF): a user, holding a secret input x, engages in a protocol with a server, holding a key sk, where at the end of the protocol the user learns $\mathsf{F}(\mathrm{sk}, \mathrm{x})$ for some keyed pseudorandom function F and the server learns nothing.

A number of primitives exist in the literature that attempt to solve this problem and provide secure and distributed key generation, however all require the storage of user-specific information with the key servers. This is infeasible at the multi-billion-user scale required for modern IM applications, and would most likely result in this feature becoming a costly paid service. In particular, securely storing this per-user key material would often be done using a hardware security module (HSM), introducing significant key management challenges and financial costs. Further, many schemes require the user to generate the high-entropy secret in the first place and then securely distribute it, imposing a trust requirement on the client device and its randomness generation. We summarize these related primitives in Sec. 4.1.2.

Problem Statement

There are at present (at least) three major stumbling blocks for deployment of encrypted backup systems for IM services:

- **Storage Cost.** Using existing techniques for OPRF-based password hardening would invoke per-user data to be stored at each of the key-contributing servers. For this reason it is necessary to minimize the storage burden for every entity in the system.
- **Key Longevity.** For the system to function, it is essential that over a long period of time the key acquisition interaction is 'long lived', in the sense that the key production operation must be deterministic and any secret values given as input to the operation (by user or server) must not change.

[2]We do not explicitly consider the scenario where the user has two devices in their possession and wishes to locally transfer messages and/or media from one device to another without the help of outsourced storage, as our approach would be overkill.

- **Trust Distribution.** In the IM setting it is by now industry standard to expect end-to-end-encryption (E2EE) of messages, ensuring that the storage server cannot decrypt sent content. With this in mind, it would appear risky to rely on the IM provider to act as a single key-contributing server, or to use only the IM server and cloud storage provider. It is thus desirable to introduce additional parties to the picture, and distribute the trust such that the user has fine-grained control of what they are able to do if they learn/believe that one or more of these parties has become compromised.

In our work we aim to overcome all of these challenges simultaneously, supplementing our proposals with rigorous security analyses.

The concept of key longevity is at odds with modern approaches to forward secrecy that involve (regular) key rotation, and so any operation that does support key rotation must also enable the user to update their ciphertexts when this rotation occurs, which is a considerable technical challenge. In Sec. 4.7 we describe how key rotation can be done securely and efficiently within our framework.

4.1.1 Contributions

We design a novel formal framework for outsourced (and on-device) storage that allows a user to generate and store a cryptographic key with the aid of n key-contributing servers, with the constraints discussed already. We call the required primitive *distributed key acquisition* (DKA) and describe its syntax and security properties. We introduce a functionality game and two key indistinguishability games for DKA. We explain how a DKA scheme can be used in conjunction with encryption and key rotation to neatly solve the IM backup problem.

We provide two concrete constructions for DKA using OPRFs in a framework that we call PERKS (Password-based Establishment of Random Keys for Storage): the n out of n setting for when the user expects the key-contributing servers to be available for the lifetime of the system, and a threshold t out of n scheme based on secret sharing that tolerates $n-t$ servers being unavailable or compromised. Even in the event that $n-t+1$ (or more, even all n) servers are corrupted by the same entity, all is not lost: this adversary must still perform an offline attack to recover the key. Our constructions are extremely simple but the analysis is certainly not: we introduce appropriate security properties for user and server privacy in our setting, and prove that our schemes—when instantiated using a variety of existing OPRFs with different features—meet the corresponding properties.

4.1.2 Related Work

The existing primitive that is most closely related to our setting is (t, n) password-protected secret sharing (PPSS): a user that is already in possession of some secret value distributes it among n servers such that reconstruction is possible using only the password and interaction with $t+1$ honest servers. Bagherzandi *et al.* [11] gave the first formal treatment and a scheme where the user needs to store/trust one or more public keys. Later schemes gave security in the presence of related passwords via the UC framework [34], and in the setting where servers learn nothing in the reconstruction phase [33] (by sending out an encryption of a randomized quotient of the password and not the password itself).

Jarecki, Kiayias and Krawczyk [58] gave threshold PPSS (and threshold PAKE) with optimal round complexity, but with the same setup assumptions as the prior papers. In [59] the same authors and Xu gained improved efficiency compared to previous work, and in essence these savings come from foregoing some heavy duty tools required to achieve the UC verifiability property of the OPRF. The same authors then presented TOPPSS [60] using a Threshold OPRF, where the secret sharing is performed on the level of the OPRF key, so derivation of the single OPRF output for a given input is an interactive protocol with $t+1$ of the n servers. Abdalla *et al.* [1] defined robust PPSS, where a robust secret sharing scheme is used to detect cheating servers, also foregoing the need for a verifiable OPRF.

Password-Hardened Encryption (PHE) was introduced by Lai *et al.* [70] and uses an oblivious external party for key derivation to protect against offline brute-force attacks on a ciphertext encrypted under just the password. Recognizing the single point of failure, threshold PHE was introduced in [31]. The scheme requires a trusted third party to distribute all secret keys during initialization.

Other primitives exist that use a distributed OPRF service (i.e. the server stores, for each user, a sharing of an OPRF secret key) to derive a cryptographic secret, including Baum *et al.* [15] who use the OPRF to get a signature key pair for distributed single sign-on, and Das *et al.* [41] who use a similar trick to obtain a signature key pair, then per-file encryption keys are created by computing a so-called extended POPRF for a second private input, namely a randomized hash of the file.

4.1.3 WhatsApp Encrypted Backup Rollout

In September 2021, WhatsApp announced [97] that they would soon begin beta testing of an encrypted chat and media backup service that uses HSMs and the envelope part of

the OPAQUE protocol [63] in a manner that is conceptually similar to a $(1,1)$-PPSS. In this subsection we discuss their system based on the details in the WhatsApp whitepaper and NCC Group's technical report [79] and explain the differences with our work.

OPAQUE is an asymmetric password-authenticated key exchange protocol that is a compiler of three components: an oblivious PRF to turn the user's password x into a strong secret value y, an 'envelope' mechanism whereby the user encrypts their AKE key material under y using symmetric encryption, and an AKE protocol. WhatsApp's approach uses the OPRF and a modified version of the envelope mechanism, but since no key exchange needs to occur the AKE component is dropped completely. In the WhatsApp system, at the point of registration (first ever backup), a client device generates a random 256-bit key k and then stores this as an encrypted record (envelope) in a 'HSM-based Backup Key Vault' so that it can later retrieve this key using only their password (PIN or passphrase): the HSM acts as the OPRF server and derives a per-user secret key sk_{uid} from a single master secret and uid when called. The envelope in the WhatsApp system is $\mathsf{PK.Enc}_{\mathrm{pk.HSM}}(\mathsf{SK.Enc}_{\mathrm{y}}(k))$, a public-key encryption of an encryption of k under the OPRF output value y. Later on when a user comes online to retrieve the contents of their envelope it is not apparent if this is sent encrypted under some user public key, and it would appear that this PKE scheme is not for protecting the channel, but rather so that the envelopes can be stored outside of the HSM. These envelopes are stored in an integrity-protected manner using a Merkle tree. No security analysis of the system has been provided for the WhatsApp approach, and the only analysis available is the report by NCC Group [79] that does not discuss any formal security requirements for the system.

Intuitively, the WhatsApp approach relies on the tamper-resistant properties of the HSM to make sure that the OPRF key sk is not leaked to any party. If this key is leaked, then an offline adversary can attempt to recover the file encryption key that is contained within the registration envelope. Our approach avoids assuming a HSM on the server side, and instead distributes the trust among a number of servers. An adversary in possession of a stolen client device needs to guess the correct password while avoiding WhatsApp's rate-limiting mechanisms, and thus performing this type of online attack is similar in our system.

Further, the WhatsApp system requires that the client device generates the user file encryption key k using 'a built-in cryptographically secure pseudorandom number generator', however as already stated, this is of no use if the device's randomness generation is already compromised during registration.

4.2 Oblivious Pseudorandom Functions

In these subsections we describe some of the properties of oblivious PRFs in the literature and explain how they can be used in our protocols.

4.2.1 OPRFs and their Variants

OPRFs can be *verifiable* or not, and independently, *partially oblivious* or not, meaning there are four categories of OPRF that we consider.

Verifiability. Verifiable OPRFs (VOPRFs) require the server to commit to the secret key that it uses, and allow the user to verify that the correct operation was performed by the server with this committed key (in a way that does not reveal the key to the user). Syntactically, the server includes a proof in in their response that the user can verify using a server public key pk.

Note that verifiability does not guarantee that a server uses the same key over multiple protocol runs. In order to check *key consistency*, the user is forced to store pk, and this value must be deterministically generated (this is the case for DH-based OPRFs where $\mathrm{pk} = g^{\mathrm{sk}}$). However, this storage need not be local: all users use the same pk so it is sufficient for this value to be published somewhere.

Partial Obliviousness. In many applications for OPRFs the server needs to partition the input space to reduce the impact of active attacks, and this is often done by choosing a different key for each user identity *uid*. In practice this could be done by applying some key derivation function to *uid* and sk before the protocol is run (see below for a short discussion of this approach). Partially-oblivious PRFs (POPRFs) contain a (plaintext) input t that provides automated partitioning, thus the server only needs one key for all users.

4.2.2 OPRF Literature

For a thorough treatment of OPRFs, see the SoK by Casacuberta *et al.* [37]; here we summarize the most important literature for our approach. Oblivious PRFs were first formally defined by Freedman *et al.* [51]. A vast array of applications has arisen for OPRFs, including oblivious transfer and private set intersection [64], secure deduplication in cloud storage [68], password-authenticated key exchange [63], Cloudflare's anonymous authentication mechanism Privacy Pass [43], checking compromised credentials [95, 75] and Meta's 'de-identified telemetry' scheme [55].

The 2HashDH scheme by Jarecki *et al.* [58] (detailed in Fig. 4.2) is very efficient and has been suggested for use in TLS 1.3 with OPAQUE as password-based authentication, and is subject to a standardization effort [42, 28, 94].

There exist generic constructions of OPRFs from MPC techniques and homomorphic encryption that do not fit into the syntax in Sec. 4.2.3 since the communication does not follow a two-message pattern with the user sending the first message, see Section 2.4 of Casacuberta *et al.* [37] for a summary. These constructions are generally useful for gaining properties that are not useful in our setting such as input batching [69] for amortized efficiency gains.

POPRFs. To our knowledge, the only two (explicit) POPRFs are those by Everspaugh *et al.* [49] and Tyagi *et al.* [96], both of which are detailed in Fig. 4.2. The former requires a pairing which could be a hurdle in some practical applications, and the latter cannot support key rotation in a straightforward way.

Three works [61, 71, 41] obtain partial-obliviousness for 2HashDH in a generic way by applying a PRF to the server key and public input and using that value as the per-user key. In our generic construction in Fig. 4.9 we use a similar idea to turn any of the two non-PO, DH-based schemes in Fig. 4.2 into partially-oblivious variants, with the additional benefit of efficient computation of per-user public keys in the verifiable setting.

The approach in the (unpublished) work of JKR18 [61] actually works for any OPRF, and they present a non-updatable construction using 2HashDH and an updatable construction that uses HashDH, which is $\mathsf{H}_2(\mathsf{H}_1(x)^{sk})$: this is not an OPRF since a user can use one interaction to obtain multiple evaluations.

Post-quantum OPRFs. Boneh *et al.* [25] gave two constructions of OPRFs from isogenies: a VOPRF from SIDH [57] with a 'one-more' assumption and an OPRF from CSIDH [38]. A year later, Basso *et al.* [14] showed that the first construction's assumption does not hold and gave attacks on that OPRF; the second CSIDH-based scheme is unaffected by this work.

From lattices, Albrecht *et al.* [3] demonstrated that it is possible to build round-optimal (two messages in the online phase) VOPRFs from the Banerjee and Peikert PRF [13], but their protocols require large parameters and computation-heavy ZK proofs.

Kolesnikov *et al.* [69] sought to build multiple concurrent OPRF operations in a generic way from oblivious transfer (OT). OT can be built from post-quantum assumptions [81, 47], however the PRF functionality requires 5 communication rounds and is only 'relaxed'

(as defined by Freedman *et al.* [51]). Note that these special purpose OPRFs, where more than two rounds are required, do not fit the syntax in Sec. 4.2.3.

Seres *et al.* [91] have proposed an OPRF based on a version of the Legendre symbol problem and linked this hardness to multivariate quadratic equation systems.

4.2.3 Syntax of Oblivious Pseudorandom Functions

Following Everspaugh *et al.* [49] and Tyagi *et al.* [96] we define OPRFs as a tuple $\mathsf{F} = (\mathsf{F.KG}, \mathsf{F.Req}, \mathsf{F.BlindEv}, \mathsf{F.Finalize}, \mathsf{F.Ev})$. The syntax captures two optional properties of OPRFs, namely partial obliviousness (user provides as input a public value in addition to its secret input, known as POPRFs) and verifiability (user is convinced that server has evaluated for a particular key, where the user learns a public commitment/representation of this key, known as VOPRFs), which both are useful but not essential in our constructions later on. In Sec. 4.2.1 we provide an overview of the prior work and how existing OPRF constructions can fit into our approach, with and without these additional properties. In Fig. 4.1 we depict the operation of a (P)OPRF with optional verifiability. Note that prior work including that of Tyagi *et al.* [96] has a parameter generation algorithm that passes some public parameters to all other algorithms. We choose to omit this detail for visual clarity.

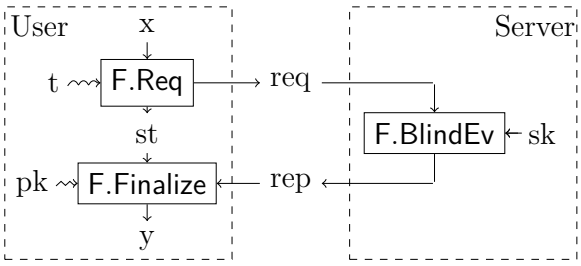

Fig. 4.1 OPRF operation diagram. Public input t is only present in POPRFs and verification public key pk is only present in VOPRFs.

$\mathsf{F.KG}$ generates a key pair $(\mathrm{pk}, \mathrm{sk})$ (or just sk for non-verifiable OPRFs). To form a request, the user runs $\mathsf{F.Req}(\mathrm{t}, \mathrm{x})$ with secret input x (and in POPRFs, public input t) and outputs a request req and a local state st. The server then runs $\mathsf{F.BlindEv}(\mathrm{sk}, \mathrm{t}, \mathrm{req})$ and outputs a response rep. The user finishes by running $\mathsf{F.Finalize}(\mathrm{pk}, \mathrm{rep}, \mathrm{st})$, either outputting the function evaluation y, or $\perp$ if it does not accept the outcome. The unblinded evaluation algorithm $\mathsf{F.Ev}(\mathrm{sk}, \mathrm{t}, \mathrm{x})$ outputs y or $\perp$. The server should not learn (anything about) the secret input x even after multiple interactions where x was provided as input, and the user should not learn (anything about) sk.

Name	POPRF?	Assumption	$\mathsf{F.Ev}(\mathrm{sk},\mathrm{t},\mathrm{x})$	req	rep
2HashDH [58]	✗	OM-Gap-DH	$\mathsf{H}_2(\mathrm{x},\mathsf{H}_1(\mathrm{x})^{\mathrm{sk}})$	$\mathsf{H}_1(\mathrm{x})^{\mathrm{r}}$	$\mathsf{H}_1(\mathrm{x})^{\mathrm{r}\cdot\mathrm{sk}}$
Pythia [49]	✓	OM-BCDH	$e(\mathsf{H}_3(\mathrm{t}),\mathsf{H}_1(\mathrm{x}))^{\mathrm{sk}}$	$\mathsf{H}_1(\mathrm{x})^{\mathrm{r}},\mathrm{t}$	$e(\mathsf{H}_3(\mathrm{t}),\mathsf{H}_1(\mathrm{x})^{\mathrm{r}})^{\mathrm{sk}}$
3HashSDHI [96]	✓	OM-Gap-SDHI	$\mathsf{H}_2(\mathrm{t},\mathrm{x},\mathsf{H}_1(\mathrm{x})^{\frac{1}{\mathsf{H}_3(\mathrm{t})+\mathrm{sk}}})$	$\mathsf{H}_1(\mathrm{x})^{\mathrm{r}},\mathrm{t}$	$\mathsf{H}_1(\mathrm{x})^{\frac{\mathrm{r}}{\mathsf{H}_3(\mathrm{t})+\mathrm{sk}}}$

Fig. 4.2 Comparison of selected (partially) oblivious PRFs from the literature. Hash functions H_i are labelled for comparison purposes, but when used in protocols the domains and ranges will be different.

In Fig. 4.2 we detail the operation of three well-known OPRF protocols, all of which are reliant on Diffie-Hellman-like assumptions. In our constructions, the key-contributing servers will hold an OPRF secret key sk that they use for all users. User separation will either be done by employing a partially oblivious PRF with $\mathrm{t} = \mathit{uid}$, or by deriving a per-user key in each protocol invocation. We will always use the password as the user's secret input, so hereon $\mathrm{x} = \mathit{pw}$.

4.2.4 Security Notions of Oblivious Pseudorandom Functions

For *correctness* we require that honest OPRF evaluations consistently produce the same output when provided with equal inputs. We formalize this requirement in the functionality game in Fig. 4.3. The adversary is granted access to the Protocol oracle which executes the protocol and records the OPRF output for each pair of public and secret input (t,x) in a query set Q. The adversary is rewarded if the size of Q becomes larger than one.

The privacy games capture that a server cannot glean any information about users' private inputs, nor link transcripts of requests/responses to OPRF output values, even with knowledge of the OPRF secret key. Our security notions are based on the security models of Tyagi *et al.* [96]. Their *user privacy* notion has two flavours:

- POPRIV-1 essentially models an honest but curious server and so does not require verifiability. In the game the adversary has a transcript-generation oracle that provides the entire transcript.
- POPRIV-2 models a malicious server by allowing the adversary to separately engage with oracles for OPRF request generation and OPRF output generation (in our notation denoted Challenge and Finalize, respectively).

Additionally they gave a pseudorandomness game for *server privacy* where an adversary interacts with a (blinded) evaluation oracle and tries to distinguish genuine operation from operation with a random function.

Game FUNC$(\mathcal{A}, n)$
00 $Q[\cdot] \leftarrow \emptyset$
01 For $i \in [1\,..\,n]$:
02 $\quad (\mathrm{pk}^i, \mathrm{sk}^i) \leftarrow_{\$} \mathsf{F.KG}$
03 $\mathcal{A}(\vec{\mathrm{pk}}, \vec{\mathrm{sk}})$
04 Lose

Oracle Protocol(t, x)
05 $(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(\mathrm{t}, \mathrm{x})$
06 For $i \in [1\,..\,n]$:
07 $\quad \mathrm{rep}^i \leftarrow \mathsf{F.BlindEv}(\mathrm{sk}^i, \mathrm{t}, \mathrm{req})$
08 $\quad \mathrm{y}^i \leftarrow \mathsf{F.Finalize}(\mathrm{pk}^i, \mathrm{rep}^i, \mathrm{st})$
09 $Q[\mathrm{t}, \mathrm{x}] \xleftarrow{\cup} \{\vec{\mathrm{y}}\}$
10 Reward $|Q[\mathrm{t}, \mathrm{x}]| > 1$
11 Return $(\vec{\mathrm{y}}, \mathrm{req}, \vec{\mathrm{rep}})$

Game PRIV-$2^b(\mathcal{A})$
12 $i \leftarrow 0$
13 $\mathrm{ST}[\cdot] \leftarrow \perp$
14 $b' \leftarrow \mathcal{A}()$
15 Stop with b'

Oracle Challenge$(\mathrm{t}, \mathrm{x}_0, \mathrm{x}_1)$
16 $i \xleftarrow{+} 1$
17 $(\mathrm{req}_0, \mathrm{st}_0) \leftarrow \mathsf{F.Req}(\mathrm{t}, \mathrm{x}_0)$
18 $(\mathrm{req}_1, \mathrm{st}_1) \leftarrow \mathsf{F.Req}(\mathrm{t}, \mathrm{x}_1)$
19 $\mathrm{ST}[i] \leftarrow (\mathrm{st}_0, \mathrm{st}_1)$
20 Return $(i, \mathrm{req}_b, \mathrm{req}_{1-b})$

Oracle Finalize$(j, \mathrm{pk}, \mathrm{rep}, \mathrm{rep}')$
21 Require $j \in [1\,..\,i]$
22 $(\mathrm{st}_0, \mathrm{st}_1) \leftarrow \mathrm{ST}[j]$
23 $\mathrm{y}_b \leftarrow \mathsf{F.Finalize}(\mathrm{pk}, \mathrm{rep}, \mathrm{st}_b)$
24 $\mathrm{y}_{1-b} \leftarrow \mathsf{F.Finalize}(\mathrm{pk}, \mathrm{rep}', \mathrm{st}_{1-b})$
25 If $\mathrm{y}_0 = \perp$ or $\mathrm{y}_1 = \perp$:
26 $\quad$ Return $(\perp, \perp)$
27 Return $(\mathrm{y}_0, \mathrm{y}_1)$

Game PRIV-$1^b(\mathcal{A})$
28 $b' \leftarrow \mathcal{A}()$
29 Stop with b'

Oracle Challenge$(\mathrm{t}, \mathrm{x}_0, \mathrm{x}_1)$
30 $(\mathrm{req}_0, \mathrm{st}_0) \leftarrow \mathsf{F.Req}(\mathrm{t}, \mathrm{x}_0)$
31 $(\mathrm{req}_1, \mathrm{st}_1) \leftarrow \mathsf{F.Req}(\mathrm{t}, \mathrm{x}_1)$
32 Return $(\mathrm{req}_b, \mathrm{req}_{1-b})$

Game PRNG$^b(\mathcal{A}, n)$
33 $\mathrm{BE}[\cdot] \leftarrow 0$; $\mathrm{RE}[\cdot] \leftarrow 0$
34 For $i \in [1\,..\,n]$:
35 $\quad (\mathrm{pk}_0^i, \mathrm{sk}_0^i) \leftarrow_{\$} \mathsf{F.KG}$
36 $\quad (\mathrm{pk}_1^i, \mathrm{sk}_1^i) \leftarrow_{\$} \mathsf{F.KG}$
37 $\quad G^i \leftarrow_{\$} \{g \mid g : \mathcal{U} \times \mathcal{I} \to \mathcal{O}\}$
38 $b' \leftarrow \mathcal{A}(\vec{\mathrm{pk}}_b)$
39 Stop with b'

Oracle Ev(t, x)
40 For $i \in [1\,..\,n]$:
41 $\quad \mathrm{y}_0 \leftarrow \mathsf{F.Ev}(\mathrm{sk}_0^i, \mathrm{t}, \mathrm{x})$
42 $\quad \mathrm{y}_1 \leftarrow G^i(\mathrm{t}, \mathrm{x})$
43 Return $\vec{\mathrm{y}}_b$

Oracle BlindEv$(\mathrm{t}, \mathrm{req})$
44 $\mathrm{BE}[\mathrm{t}] \xleftarrow{+} 1$
45 For $i \in [1\,..\,n]$:
46 $\quad \mathrm{rep}_0^i \leftarrow \mathsf{F.BlindEv}(\mathrm{sk}_0^i, \mathrm{t}, \mathrm{req})$
47 $\quad \mathrm{rep}_1^i \leftarrow \mathcal{S}.\mathsf{BlindEv}(\mathrm{sk}_1^i, \mathrm{t}, \mathrm{req})$
48 Return $\vec{\mathrm{rep}}_b$

Oracle $H(x)$
49 $h_0 \leftarrow \mathsf{RO}(x)$
50 $h_1 \leftarrow \mathcal{S}.\mathsf{Ev}(x)$
51 Return h_b

$\mathcal{S}$**-Oracle** RestrictedEv(t, x)
52 Require $\mathrm{RE}[\mathrm{t}] < \mathrm{BE}[\mathrm{t}]$
53 $\mathrm{RE}[\mathrm{t}] \xleftarrow{+} 1$
54 $\vec{\mathrm{y}} \leftarrow \mathrm{Ev}(\mathrm{t}, \mathrm{x})$
55 Return $\vec{\mathrm{y}}$

Fig. 4.3 OPRF Games for the multiple servers setting. For the meaning of instructions Stop with, Lose, Reward, and Require see Sec. 1.3.

Our PRIV-x games are adapted from the POPRIV-x games of Tyagi *et al.* [96]; in Sec. 4.2.5 we detail the differences. In short, PRIV-1 is a simplification of POPRIV-1 but with equivalent security, while PRIV-2 and POPRIV-2 are identical.

We remark we only return the request req and not the response rep nor the final output value y in PRIV-1. It should be obvious that the adversary can compute rep on its own because it holds the secret key to evaluate $\mathsf{F.BlindEv}$. (In fact it can run this operation with arbitrary secret keys.) Therefore, by functionality, the adversary can obtain y by generating the request itself using the same *pw* as submitted to the Challenge oracle, compute the response rep and run the $\mathsf{F.Finalize}$ algorithm to compute y and obtain a full transcript. In PRIV-2 we do have a Finalize oracle, as the adversary is allowed to submit arbitrary responses and the functionality game makes no statement about this case. In Sec. 4.2.5 we show that PRIV-1 and POPRIV-1 are equivalent. Furthermore, the $\mathsf{F.Req}$ algorithm no longer takes a public key as input and is thus independent of the server. Instead, the $\mathsf{F.Finalize}$ algorithm, which uses the public key to verify the response, now takes in the public key directly, rather than it being passed via request state.

Definition 4.2.1 (PRIV-x Security)**.** *The advantage of an adversary* $\mathcal{A}$ *in the* PRIV-x *security games defined in Fig. 4.3 for* $\mathrm{x} \in \{1,2\}$ *and OPRF* F *is*

$$\mathbf{Adv}_{\mathsf{F}}^{\text{PRIV-x}}(\mathcal{A}) := \left|\mathbb{P}\left[\mathsf{G}_{\mathsf{F}}^{\text{PRIV-x}^1}(\mathcal{A}) = 1\right] - \mathbb{P}\left[\mathsf{G}_{\mathsf{F}}^{\text{PRIV-x}^0}(\mathcal{A}) = 1\right]\right|.$$

Our PRNG^b game is similar to the POPRF game in [96] for verifiable POPRFs, but extended to our multi-server setting: the oracles now return vectors instead of single elements. The game is parameterized by a simulator $\mathcal{S}$ and creates an environment for an adversary to interact with n OPRF servers via an oracle for function evaluation (Ev) and in modelling malicious clients a blinded evaluation oracle (BlindEv). Initially, the game generates 2n server key pairs (or just secret keys for OPRFs that are not verifiable): in the $b = 0$ case the real function is used with one of the secret keys, and in the $b = 1$ case a random function is used and the simulator is tasked with providing appropriate responses but given the other secret key. The Ev oracle returns either the output of the $\mathsf{F.Ev}$ algorithm or a random function. The adversary also has access to the BlindEv oracle, which either returns the output of the $\mathsf{F.BlindEv}$ algorithm or the response generated by the simulator $\mathcal{S}.\mathsf{BlindEv}$. Queries to oracle H are either answered by a random oracle query or simulated by $\mathcal{S}.\mathsf{Ev}$. Crucially, to maintain consistency between Ev and BlindEv queries, the simulator can obtain Ev outputs via its own $\mathcal{S}$-Oracle RestrictedEv. However, the simulator is restricted: the number of queries to

RestrictedEv is bounded by the number of adversary queries to BlindEv, specific for each public input. This ensures that the adversary cannot compute more POPRF evaluations than the number of oracle queries it made. Moreover, restricting per public input means that querying with public input t_1 cannot help the adversary compute the evaluation for another public input $\mathrm{t}_2 \neq \mathrm{t}_1$. For a more detailed description of (the single server version of) this game motivating the modelling choices, see Section 3 of [96]: there are many subtleties discussed regarding the simulator and limited evaluation. For our purposes, it is sufficient to know we only build from secure (P)OPRFs, that is an OPRF for which all realistic adversaries have negligible advantage in the security games defined in Fig. 4.3.

Definition 4.2.2 (PRNG Security)**.** *The advantage of an adversary $\mathcal{A}$ in the* PRNG *security game defined in Fig. 4.3 for OPRF* F *is*

$$\mathbf{Adv}^{\mathrm{PRNG}}_{\mathsf{F},\mathcal{S}}(\mathcal{A},\mathrm{n}) := \left|\mathbb{P}\left[\mathsf{G}^{\mathrm{PRNG}^1}_{\mathsf{F},\mathcal{S}}(\mathcal{A},\mathrm{n}) = 1\right] - \mathbb{P}\left[\mathsf{G}^{\mathrm{PRNG}^0}_{\mathsf{F},\mathcal{S}}(\mathcal{A},\mathrm{n}) = 1\right]\right|.$$

4.2.5 OPRF Definition Relations

Comparison between Mechanics of PRIV-1 and POPRIV-1

We now summarize the differences between the POPRIV-1 game of Tyagi *et al.* [96] given in our notation in Fig. 4.4 and our PRIV-1 notion given in Fig. 4.3 (recall that our PRIV-2 game is identical to the POPRIV-2 game).

Game POPRIV-$1^b(\mathcal{A})$	**Oracle** TRANS$(uid, \mathrm{x}_0, \mathrm{x}_1)$
00 $(\mathrm{pk},\mathrm{sk}) \leftarrow_{\$} \mathsf{F.KG}$	03 $(\mathrm{req}_0, \mathrm{st}_0) \leftarrow \mathsf{F.Req}(\mathrm{t}, \mathrm{x}_0)$
01 $b' \leftarrow \mathcal{A}(\mathrm{pk},\mathrm{sk})$	04 $(\mathrm{req}_1, \mathrm{st}_1) \leftarrow \mathsf{F.Req}(\mathrm{t}, \mathrm{x}_1)$
02 Stop with b'	05 $\mathrm{rep}_0 \leftarrow \mathsf{F.BlindEv}(\mathrm{sk}, \mathrm{t}, \mathrm{req}_0)$
	06 $\mathrm{rep}_1 \leftarrow \mathsf{F.BlindEv}(\mathrm{sk}, \mathrm{t}, \mathrm{req}_1)$
	07 $\mathrm{y}_0 \leftarrow \mathsf{F.Finalize}(\mathrm{pk}, \mathrm{rep}_0, \mathrm{st}_0)$
	08 $\mathrm{y}_1 \leftarrow \mathsf{F.Finalize}(\mathrm{pk}, \mathrm{rep}_1, \mathrm{st}_1)$
	09 $\tau \leftarrow (\mathrm{req}_b, \mathrm{rep}_b, \mathrm{y}_0)$
	10 $\tau \leftarrow (\mathrm{req}_{1-b}, \mathrm{rep}_{1-b}, \mathrm{y}_1)$
	11 Return (τ, τ')

Fig. 4.4 POPRIV-1 security game [96].

In the POPRIV-1 game the adversary receives an OPRF key pair and access to a transcript oracle TRANS that runs the server side of the OPRF functionality using inputs $(uid, \mathrm{x}_0, \mathrm{x}_1)$ given by the adversary, and returns the request and response values in random order. In our PRIV-1 game the adversary is not given a key pair but has access to a challenge oracle Challenge with the same inputs $(uid, \mathrm{x}_0, \mathrm{x}_1)$, but only receives

$(\text{req}_b, \text{req}_{1-b})$. Our observation is that the adversary can act as the server, i.e. running the deterministic process F.BlindEv, for arbitrary key pairs. Recall that we give public key pk as an input to F.Finalize rather than embedding it in the state value st as done by [96].

PRIV-x and POPRIV-x are Equivalent

We now show the equivalence of our multi-server PRIV-x games and the single-server POPRIV-x games introduced by Tyagi *et al.* [96]. As stated earlier, our PRIV-2 game is identical to the POPRIV-2 game of Tyagi *et al.*, and so we focus on showing PRIV-1 $\Leftrightarrow$ POPRIV-1.

Theorem 4.2.1. *Let* F *be an oblivious pseudorandom function. For any adversary* $\mathcal{A}$ *against the* PRIV-1 *security of* F*, there exists an adversary* $\mathcal{B}$ *against the* POPRIV-1 *security of* F*, such that*

$$\mathbf{Adv}_{\mathsf{F}}^{\text{PRIV-1}}(\mathcal{A}) \leq \mathbf{Adv}_{\mathsf{F}}^{\text{POPRIV-1}}(\mathcal{B}).$$

Proof. The direct reduction is detailed in Fig. 4.5. $\mathcal{B}$ runs $\mathcal{A}$, and needs to respond to $\mathcal{A}$'s calls to Challenge. Note that $\mathcal{A}$'s calls to Challenge give $(uid, \text{x}_0, \text{x}_1)$ as input, and $\mathcal{A}$ expects $(\text{req}_b, \text{req}_{1-b})$ in response, when it is playing PRIV-1^b. $\mathcal{B}$'s own oracle $\text{TRANS}_{\mathcal{B}}$ provides a more detailed response, and so $\mathcal{B}$ simply takes the $\text{req}_b, \text{req}_{1-b}$ that it receives and forwards this to $\mathcal{A}$.

Let b be the challenge bit in the game that $\mathcal{B}$ is playing, and let b' be the bit that is output by $\mathcal{A}$. $\mathcal{B}$ receives $(\text{req}_b, \text{rep}_b, \text{y}_0, \text{req}_{1-b}, \text{rep}_{1-b}, \text{y}_1)$ from its own call to $\text{TRANS}_{\mathcal{B}}$, and thus providing $(\text{req}_b, \text{req}_{1-b})$ to $\mathcal{A}$ simulates $\mathcal{A}$'s expected environment.

$\mathcal{B}$ perfectly simulates PRIV-1^b for $\mathcal{A}$, since the secret key vector is correctly distributed and the responses that $\mathcal{A}$ receives to its oracles calls are exactly as it would expect. The advantage of $\mathcal{A}$ directly corresponds to the advantage of $\mathcal{B}$. This concludes the proof.

Reduction $\mathcal{B}$ **playing** POPRIV-1^b	**Oracle** $\text{Challenge}_{\mathcal{A}}(uid, \text{x}_0, \text{x}_1)$
00 **receive** pk, sk	03 **call** $\text{TRANS}_{\mathcal{B}}(uid, \text{x}_0, \text{x}_1)$
01 $b' \leftarrow \mathcal{A}()$	04 **receive** $(\text{req}_b, \text{rep}_b, \text{y}_0, \text{req}_{1-b}, \text{rep}_{1-b}, \text{y}_1)$
02 Return b'	05 Return $(\text{req}_b, \text{req}_{1-b})$

Fig. 4.5 Reduction $\mathcal{B}$ for the proof of Thm. 4.2.1.

□

Theorem 4.2.2. *Let* F *be an oblivious pseudorandom function. For any adversary* $\mathcal{A}$ *against the* POPRIV-1 *security of* F*, there exists an adversary* $\mathcal{B}$ *against the* PRIV-1 *security of* F*, such that*

$$\mathbf{Adv}_{\mathsf{F}}^{\text{POPRIV-1}}(\mathcal{A}) \leq \mathbf{Adv}_{\mathsf{F}}^{\text{PRIV-1}}(\mathcal{B}).$$

Proof. The reduction is detailed in Fig. 4.6. In order to provide a sufficient response to

Reduction $\mathcal{B}$ **playing** PRIV-1^b	**Oracle** $\text{TRANS}_{\mathcal{A}}(uid, \text{x}_0, \text{x}_1)$
00 $(\text{pk}, \text{sk}) \leftarrow \mathsf{F.KG}$	03 **call** $\text{Challenge}_{\mathcal{B}}(uid, \text{x}_0, \text{x}_1)$
01 $b' \leftarrow \mathcal{A}(\text{pk}, \text{sk})$	04 **receive** $(\text{req}_b, \text{req}_{1-b})$
02 Return b'	05 $\text{rep}_b \leftarrow \mathsf{F.BlindEv}(\text{sk}, uid, \text{req}_b)$
	06 $\text{rep}_{1-b} \leftarrow \mathsf{F.BlindEv}(\text{sk}, uid, \text{req}_{1-b})$
	07 $\text{y}_0 \leftarrow \mathsf{F.Ev}(\text{sk}, uid, \text{x}_0)$
	08 $\text{y}_1 \leftarrow \mathsf{F.Ev}(\text{sk}, uid, \text{x}_1)$
	09 Return $(\text{req}_b, \text{rep}_b, \text{y}_0, \text{req}_{1-b}, \text{rep}_{1-b}, \text{y}_1)$

Fig. 4.6 Reduction $\mathcal{B}$ for the proof of Thm. 4.2.2.

$\mathcal{A}$, the reduction $\mathcal{B}$ must use the values $(\text{req}_b, \text{req}_{1-b})$ that it receives from $\text{TRANS}_{\mathcal{A}}$ and perform $\mathsf{F.BlindEv}$ on them to acquire $(\text{rep}_b, \text{rep}_{1-b})$. Producing y_0 and y_1 is straightforward, since $\mathcal{B}$ can simply compute the OPRF evaluation with sk_j and the input values x_0 and x_1. $\mathcal{B}$ combines the values into output $(\text{req}_b, \text{rep}_b, \text{y}_0, \text{req}_{1-b}, \text{rep}_{1-b}, \text{y}_1)$ and returns it to $\mathcal{A}$. The reduction perfectly simulates the POPRIV-1^b environment for $\mathcal{A}$. This concludes the proof. □

4.3 DKA and Security Models

In this section we formally define the syntax of a distributed key acquisition scheme and define security via two games for key indistinguishability. The weaker KIND-1 security game effectively models an honest but curious adversary as it may call oracles for honest protocol executions for a user to learn its requests and responses. In the KIND-2 security game the adversary is in complete control of the servers and may choose arbitrarily how to respond to user requests.

4.3.1 Distributed Key Acquisition

A distributed key acquisition scheme is an interactive protocol between parties, where the parties can either be users or servers.

SYNTAX. A distributed key acquisition scheme for a set S of n servers that are (initially) available to users consists of a secret key space $\mathcal{SK}$, a public key space $\mathcal{PK}$, a user identity space $\mathcal{UID}$, a dictionary $\mathcal{D}$, a key space $\mathcal{K}$, an output space $\mathcal{O}$, algorithms gen, init, acquire, recover and server.op.

The system is initialized by gen which assigns a key pair $(\mathrm{sk}, \mathrm{pk}) \in \mathcal{SK} \times \mathcal{PK}$ to each server $\mathrm{S}_i \in \mathrm{S}$, and public keys $\vec{\mathrm{pk}} \in \mathcal{PK}^n$ are then distributed to users. A user $uid \in \mathcal{UID}$ with password $pw \in \mathcal{D}$ initializes itself in the system and acquires a key $k \in \mathcal{K}$ and possibly some setup values $\vec{\mathrm{SV}} \in \mathcal{O}^n$ by running $\mathrm{init}(uid, pw, \vec{\mathrm{pk}}, \mathrm{S})$. We remark that $\vec{\mathrm{SV}}$ is not secret, so it can be stored alongside the backed up data. The init procedure sends out a request $\mathrm{req} \in \mathcal{RQ}$ to each server, who will run $\mathsf{server.op}(\mathrm{sk}_i, uid, \mathrm{req})$ to respond with response $\mathrm{rep} \in \mathcal{RP}$. An individual user's interaction with the system is defined by a threshold $\mathrm{t} \in [1 \,..\, \mathrm{n}]$ which is the number of servers that are required to be honest and available in order for the user to reconstruct their secret. Later, a user can acquire output values $\vec{y} \in \mathcal{O}^n$ by running $\mathrm{acquire}(uid, pw, \vec{\mathrm{pk}}, \mathrm{S})$ and recover their key $k \in \mathcal{K}$ by subsequently running $\mathrm{recover}(\vec{y}, \mathrm{C}, \vec{\mathrm{SV}})$, where the set of chosen servers C is a subset of S. Syntactically this takes as input all output values, but some may not be set, i.e. y_i may be $\perp$ if server S_i did not respond. With C, a user can indicate which output values to use for the key recovery. Thus, if $\mathcal{P}(\mathrm{S})$ denotes the powerset of S, the distributed key acquisition API is as follows:

$$\mathrm{gen} \to (\mathcal{SK} \times \mathcal{PK})^{\mathrm{n}} \qquad \mathcal{UID} \times \mathcal{D} \times \mathcal{PK}^n \times \{\mathrm{S}\} \to \mathrm{init} \to \mathcal{K} \times \mathcal{O}^n$$

$$\mathcal{SK} \times \mathcal{UID} \times \mathcal{RQ} \to \mathsf{server.op} \to \mathcal{RP}$$

$$\mathcal{UID} \times \mathcal{D} \times \mathcal{PK}^n \times \{\mathrm{S}\} \to \mathrm{acquire} \to \mathcal{O}^n \qquad \mathcal{O}^n \times \mathcal{P}(\mathrm{S}) \times \mathcal{O}^n \to \mathrm{recover} \to \mathcal{K}$$

We assume that passwords are selected uniformly at random from the set $\mathcal{D}$ throughout the rest of this chapter. However, similarly to the game-based PAKE literature, it is possible to cast the choosing of passwords according to (the min-entropy of) some distribution Dist [30, 20].

4.3.2 A Unified Security Notion for DKA

We first describe the functionality game for DKA schemes depicted in Fig. 4.7. The functionality game initializes the secret and public keys for all servers and initializes several game variables to keep track of the game state. The adversary controls the honest executions of the protocol via its oracles Init, Acquire and Recover. It has complete control over the inputs, specifying which user identity *uid*, password *pw* and public keys

Game FUNC($\mathcal{A}$, t, n)
00 $\mathrm{K}[\cdot] \leftarrow \emptyset$
01 $\mathrm{CRED}[\cdot] \leftarrow \perp$
02 $\mathrm{Y}[\cdot] \leftarrow \perp$
03 $\mathrm{CO} \leftarrow \emptyset$
04 $r \leftarrow 0$
05 $(\vec{\mathrm{sk}}, \vec{\mathrm{pk}}) \leftarrow \mathrm{gen}$
06 $\mathrm{S} \leftarrow \mathrm{init}(\vec{\mathrm{sk}})$
07 $\mathcal{A}(\vec{\mathrm{pk}})$
08 Lose

Oracle Init($uid, pw, \vec{\mathrm{pk}}$)
09 $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{init}(uid, pw, \vec{\mathrm{pk}}, \mathrm{S})$
10 $\mathrm{K}[pw, uid, \vec{\mathrm{SV}}] \xleftarrow{\cup} \{k\}$
11 $(\mathrm{req}, \vec{\mathrm{rep}}) \leftarrow \mathrm{Transcript}(\mathrm{S})$
12 Return $(k, \mathrm{req}, \vec{\mathrm{rep}}, \vec{\mathrm{SV}})$

Oracle Acquire($uid, pw, \vec{\mathrm{pk}}$)
13 $r \xleftarrow{+} 1$
14 $\mathrm{CRED}[r] \leftarrow (uid, pw)$
15 $\mathrm{Y}[r] \leftarrow \mathrm{acquire}(uid, pw, \vec{\mathrm{pk}}, \mathrm{S})$
16 $(\mathrm{req}, \vec{\mathrm{rep}}) \leftarrow \mathrm{Transcript}(\mathrm{S})$
17 Return $(\mathrm{req}, \vec{\mathrm{rep}}, r)$

Oracle Corrupt(i)
18 $\mathrm{CO} \xleftarrow{\cup} \{i\}$
19 Require $|\mathrm{CO}| < \mathrm{t}$
20 Return sk_i

Oracle Recover($r, \mathrm{C}, \vec{\mathrm{SV}}$)
21 $(uid, pw) \leftarrow \mathrm{CRED}[r]$
22 $\vec{\mathrm{y}} \leftarrow \mathrm{Y}[r]$
23 $k \leftarrow \mathrm{recover}(\vec{\mathrm{y}}, \mathrm{C}, \vec{\mathrm{SV}})$
24 $\mathrm{K}[pw, uid, \vec{\mathrm{SV}}] \xleftarrow{\cup} \{k\}$
25 Reward $|\mathrm{K}[pw, uid, \vec{\mathrm{SV}}]| > 1$
26 Return k

Fig. 4.7 Functionality game for DKA with algorithms gen, init, acquire and recover. Transcript is a special game procedure that records network requests sent by the DKA algorithms. For the meaning of instructions Lose, Reward, and Require see Sec. 1.3.

$\vec{\mathrm{pk}}$ to use in Init and Acquire, and the set of chosen servers C (for reconstruction) and setup values $\vec{\mathrm{SV}}$ in the Recover oracle. The counter r is a game variable to associate the Recover query with the corresponding Acquire query. This gives the adversary more freedom as we do not require these oracles to be used in succession. Via the Corrupt oracle the adversary is allowed to corrupt up to $t-1$ servers. Recall that the oracle immediately aborts if a procedure aborts. In particular, a correct construction can abort if it is fed garbage input (i.e. an empty set of chosen servers) as otherwise the adversary would trigger the 'Reward' line that wins the functionality game. The adversary wins the functionality game if it manages to create two different keys for a set of user id, password and setup values.

Next, we describe the security games for DKA schemes, KIND-x for $\mathrm{x} \in \{1, 2\}$, provided in Fig. 4.8, which ultimately capture key indistinguishability: The task of the adversary is to distinguish real keys generated by the protocol from random. We remark that this implies other security properties such as privacy of the user's password: if the adversary learns (information about) a user's password it can compute the real key and compare this to the real or random key from the Challenge oracle to gain an advantage. Similarly to PRIV-x, the KIND-x game comes in two flavours: $\mathrm{x} = 1$ corresponds to an adversary that may compromise servers but will subsequently follow the protocol honestly, and $\mathrm{x} = 2$ which allows arbitrary server behaviour and thus intuitively security in this setting will require verifiable responses from the servers.

Initially, the game assigns passwords to all users in the user identity space $\mathcal{UID}$. An adversary can observe network traffic for executions of the protocol via its oracles Init

and Acquire, and it specifies the user identity for these protocol runs. In the KIND-1 game for honest executions of the protocol, this is modelled by providing the adversary a transcript of the network requests to the servers S, handled by the game. For the KIND-2 game, the network requests are sent to the adversary directly, so the game does not need to record the transcript. The adversary may respond in any way it likes, in particular it may respond honestly. To aid the adversary in responding honestly without requiring it to corrupt a server, it can query the BlindEv oracle, which will return the honest response rep. Effectively, in the KIND-2 game the adversary becomes an active man-in-the-middle between the Init and Acquire oracles (representing the user) and the BlindEv oracle (representing the server).

We split the key reconstruction procedure into two processes to model the online communication (Acquire) between the user and the servers, and the key calculation done locally (Recover) by a user based on the servers it chooses to utilize and the public setup values. We allow the adversary to specify these setup values $\vec{\mathrm{SV}}$ since the system's security should not rely on them being secret nor authentic. To model online attacks, i.e. login attempts for specific users, the adversary can call Reveal with a purported password and user identity, and if this guess is correct it receives the file encryption key for that user; if incorrect it receives nothing (this mimics the subsequent inability to decrypt files).

Definition 4.3.1 (KIND-x Security)**.** *The advantage of an adversary* $\mathcal{A}$ *in the* KIND-x *games defined in Fig. 4.8 for* $\mathrm{x} \in \{1,2\}$ *and distributed key acquisition scheme* DKA *is*

$$\mathbf{Adv}^{\text{KIND-x}}_{\mathsf{DKA}}(\mathcal{A}) := \left| \mathbb{P}\left[\mathsf{G}^{\text{KIND-x}^1}_{\mathsf{DKA}}(\mathcal{A}) = 1 \right] - \mathbb{P}\left[\mathsf{G}^{\text{KIND-x}^0}_{\mathsf{DKA}}(\mathcal{A}) = 1 \right] \right|.$$

As is natural in the password setting, it is necessary to consider the fact that passwords could be guessable and a successful guess that occurs before any rate-limiting has kicked in will result in an adversary compromising a particular user. The advantage statement needs to take into account the following generic attack, where the queries are all for a single user identity *uid*: the adversary runs Init, then makes q queries to Reveal for randomly chosen passwords in the password space, then queries Challenge. If any of the Reveal queries returned something other than $\perp$, then a user key was set for that user identity, and if this value is equal to the key provided by Challenge then the adversary outputs 0, and if it's different it outputs 1 (if it received $\perp$ for all Reveal queries then it just guesses). As a result, we regard a DKA scheme as being secure if

$$\mathbf{Adv}^{\text{KIND-x}}_{\mathsf{DKA}}(\mathcal{A}) \leq \mathcal{O}\left(\frac{\mathrm{q}}{|\mathcal{D}|}\right) + \delta,$$

Game KIND-1$^b(\mathcal{A}, \mathrm{t}, \mathrm{n})$
00 $\mathrm{K}[\cdot] \leftarrow \times$; $r \leftarrow 0$
01 $\mathrm{PW}[\cdot] \leftarrow \perp$; $\mathrm{Y}[\cdot] \leftarrow \perp$
02 For $uid \in \mathcal{UID}$:
03 $\quad \mathrm{PW}[uid] \leftarrow_{\$} \mathcal{D}$
04 $\mathrm{CH} \leftarrow \emptyset$; $\mathrm{CO} \leftarrow \emptyset$
05 $(\vec{\mathrm{sk}}, \vec{\mathrm{pk}}) \leftarrow$ gen
06 $\mathrm{S} \leftarrow \mathrm{init}(\vec{\mathrm{sk}})$
07 $b' \leftarrow \mathcal{A}(\vec{\mathrm{pk}})$
08 Stop with b'

Oracle Init$(uid, \vec{\mathrm{pk}})$
09 $pw \leftarrow \mathrm{PW}[uid]$
10 $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{init}(uid, pw, \vec{\mathrm{pk}}, \mathrm{S})$
11 $\mathrm{K}[pw, uid] \leftarrow k$
12 $(\mathrm{req}, \vec{\mathrm{rep}}) \leftarrow \mathrm{Transcript}(\mathrm{S})$
13 Return $(\mathrm{req}, \vec{\mathrm{rep}}, \vec{\mathrm{SV}})$

Oracle Acquire$(uid, \vec{\mathrm{pk}})$
14 $r \xleftarrow{+} 1$
15 $pw \leftarrow \mathrm{PW}[uid]$
16 $\mathrm{Y}[r] \leftarrow \mathrm{acquire}(uid, pw, \vec{\mathrm{pk}}, \mathrm{S})$
17 $(\mathrm{req}, \vec{\mathrm{rep}}) \leftarrow \mathrm{Transcript}(\mathrm{S})$
18 Return $(\mathrm{req}, \vec{\mathrm{rep}}, r)$

Oracle Recover$(r, \mathrm{C}, \vec{\mathrm{SV}})$
19 $\vec{\mathrm{y}} \leftarrow \mathrm{Y}[r]$
20 $k \leftarrow \mathrm{recover}(\vec{\mathrm{y}}, \mathrm{C}, \vec{\mathrm{SV}})$
21 $\mathrm{K}[pw, uid] \leftarrow k$
22 Return

Oracle Reveal(pw', uid)
23 $k \leftarrow \mathrm{K}[pw', uid]$
24 Return k

Oracle Challenge(uid)
25 $pw \leftarrow \mathrm{PW}[uid]$
26 Require $\mathrm{K}[pw, uid] \neq \times$
27 Require $uid \notin \mathrm{CH}$
28 $\mathrm{CH} \xleftarrow{\cup} \{uid\}$
29 $k_0 \leftarrow \mathrm{K}[pw, uid]$
30 $k_1 \leftarrow_{\$} \mathcal{K}$
31 Return k_b

Oracle Corrupt(i)
32 $\mathrm{CO} \xleftarrow{\cup} \{i\}$
33 Require $|\mathrm{CO}| < \mathrm{t}$
34 Return sk_i

Game KIND-2$^b(\mathcal{A}, \mathrm{t}, \mathrm{n})$
35 $\mathrm{K}[\cdot] \leftarrow \times$; $r \leftarrow 0$
36 $\mathrm{PW}[\cdot] \leftarrow \perp$ $\mathrm{Y}[\cdot] \leftarrow \perp$
37 For $uid \in \mathcal{UID}$:
38 $\quad \mathrm{PW}[uid] \leftarrow_{\$} \mathcal{D}$
39 $\mathrm{CH} \leftarrow \emptyset$; $\mathrm{CO} \leftarrow \emptyset$
40 $(\vec{\mathrm{sk}}, \vec{\mathrm{pk}}) \leftarrow$ gen
41 $b' \leftarrow \mathcal{A}(\vec{\mathrm{pk}})$
42 Stop with b'

Oracle BlindEv(i, uid, req)
43 $\mathrm{rep}_i \leftarrow \mathsf{server.op}(\mathrm{sk}_i, uid, \mathrm{req})$
44 Return rep_i

Oracle Init$(uid, \vec{\mathrm{pk}})$
45 $pw \leftarrow \mathrm{PW}[uid]$
46 $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{init}(uid, pw, \vec{\mathrm{pk}}, \mathcal{A})$
47 $\mathrm{K}[pw, uid] \leftarrow k$
48 Return $\vec{\mathrm{SV}}$

Oracle Acquire$(uid, \vec{\mathrm{pk}})$
49 $r \xleftarrow{+} 1$
50 $pw \leftarrow \mathrm{PW}[uid]$
51 $\mathrm{Y}[r] \leftarrow \mathrm{acquire}(uid, pw, \vec{\mathrm{pk}}, \mathcal{A})$
52 Return r

Fig. 4.8 Key indistinguishability games for DKA with algorithms gen, init, acquire, recover and server.op. The oracles in the middle column are equal for both games and hence only displayed once. Transcript is a special game procedure that records network requests sent by the DKA algorithms. Assuming $\times \notin \mathcal{K}$, we encode uninitialized keys with $\times$. For the meaning of instructions Stop with and Require see Sec. 1.3.

where q is the number of queries made by the adversary to the Reveal oracle in the course of the game, $|\mathcal{D}|$ is the size of the password dictionary and δ is some negligible function in the security parameter.

4.4 Constructions

In this section we present two schemes, parameterized by an Oblivious PRF $\mathsf{F} = (\mathsf{F.KG}, \mathsf{F.Req}, \mathsf{F.BlindEv}, \mathsf{F.Finalize}, \mathsf{F.Ev})$, and prove their security in the models from Sec. 4.3. Our constructions follow a generic blueprint, portrayed in Fig. 4.9.

4.4.1 Generic Construction

There are four possibilities for OPRFs: verifiable or non-verifiable, and partially-oblivious or regular. In order to handle both partially-oblivious and regular oblivious PRFs, we desire that each server can derive a per-user key on the fly, see the Cli.SKG and Cli.PKG algorithms in Fig. 4.9. If the OPRF is partially oblivious then the auxiliary input *uid* creates domain separation in the key used by the OPRF server, and so the per-user key pair is just the single key pair created by F.KG, for all users. To work with verifiable (regular) OPRFs such that the servers are not required to store per-user data, we need the OPRF secret key sk to be a group element with public key g^{sk} for some generator g. Then, the server simply multiplies in the group its own OPRF secret key with a hash of the user's identity to create the secret key component, and raises its own public key to the hash value to get the public key component; this public key component is provided to the user. If a verifiable OPRF is being used and it does not have this method of operation, and this includes all OPRFs built from non-DH assumptions, then another mechanism is required. For non-verifiable (non-DH) OPRFs, the server simply needs some way of generating per-user secret keys using its master secret key and the user's identity, e.g. a key derivation function.

The init algorithm allows the user to compute a random key using its password *pw* and a set of servers S that can be reconstructed later using *pw* and (a subset of) S. It creates an OPRF request and awaits the response for each server. We have used the 'Await' keyword in Fig. 4.9 to indicate this computation is not done locally. Computing the OPRF responses is done by server.op, which is a wrapper of the OPRF's blind evaluation function using the per-user key. The responses are finalized by init to obtain the OPRF output values, which are used by setup to compute the key and potentially some setup values. The setup algorithm is setting specific and will be discussed later.

The reconstruction of the key is similar, but split in two algorithms acquire and recover. This allows the user to choose which subset of servers C to use, after seeing the OPRF outputs (some servers may not respond). Indeed, this modularization allows any choice function from the OPRF output space to the power set of S. The OPRF output values are computed by acquire, to be used subsequently by the local reconstruct algorithm inside recover to recompute the key. Similarly to setup, the reconstruct algorithm is setting specific.

We remark that the scheme stops working (in the sense that the user cannot decrypt their files) if one of the servers chosen for recover is not consistent with its responses. If the OPRF is verifiable, then the user can identify which server has replied inconsistently and exclude it from the servers chosen for recover, so it is recommended to use verifiable

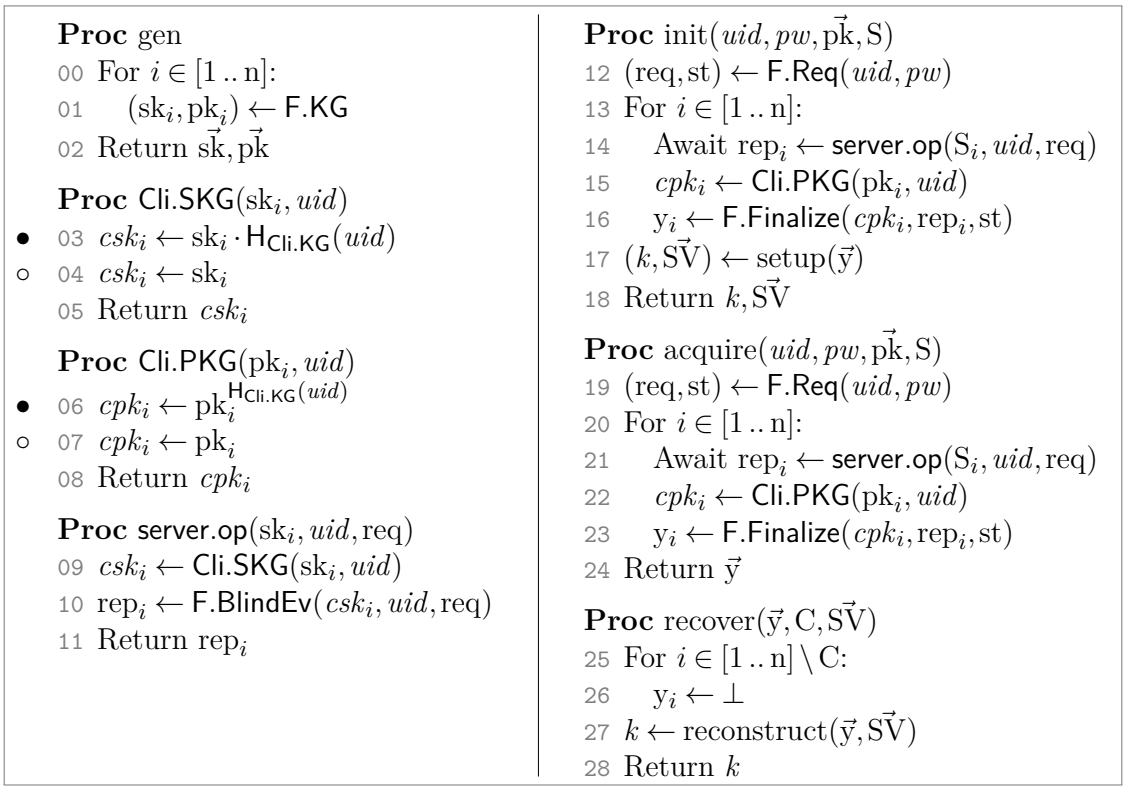

Fig. 4.9 PERKS, a generic DKA protocol construction. The lines marked with • are executed iff a standard OPRF is used as building block. The lines marked with ○ are executed iff a POPRF is used as building block. Procedures setup and reconstruct are as in Fig. 4.10 for the n out of n setting and as in Fig. 4.11 for the t out of n setting. In a slight abuse of notation, we specify server.op on the user side with a server S_i as input, who will use its secret key sk_i to evaluate the procedure.

OPRFs when available. Alternatively, assuming the set of servers is small, the user could proceed by trying different subsets and rerunning recover until successful. In Sec. 4.6 we discuss how existing OPRF schemes from the literature can be used in PERKS.

4.4.2 n out of n setting

In Fig. 4.10 we provide the construction for the setting where all n key servers are required for key (re)production. The setup values $\vec{SV}$ are effectively ignored in this setting, they are only present in the construction to be syntactically correct. The user derives their key as the XOR of the OPRF output values. The construction allows a user to generate a key even if it cannot produce randomness itself, and as long as at least one of the n servers is not malicious, the key produced will be pseudorandom.

Proc setup($\vec{y}$)	**Proc** reconstruct($\vec{z}, \vec{SV} = \perp$)
00 Assert size($\vec{y}$) = n	03 Assert size($\vec{z}$) = n
01 $k \leftarrow y_1 \oplus \ldots \oplus y_n$	04 $k \leftarrow z_1 \oplus \ldots \oplus z_n$
02 Return $(k, \vec{SV} = \perp)$	05 Return k

Fig. 4.10 Construction for n out of n setting.

Proc setup($\vec{y}$)	**Proc** reconstruct($\vec{z}, \vec{SV}$)
00 Assert size($\vec{y}$) = n	05 Assert size($\vec{z}$) = t
01 $k \leftarrow_{\$} \mathcal{K}$	06 $S = \emptyset$
02 $\vec{\alpha} \leftarrow$ SecShare(k, t, n)	07 For each $z_i \neq \perp$:
03 $\vec{SV} \leftarrow \vec{\alpha} + \vec{y}$	08 $\quad \alpha_i \leftarrow \vec{SV}_i - z_i$
04 Return $k, \vec{SV}$	09 $\quad \vec{\alpha}' \xleftarrow{\cup} \{\alpha_i\}$
	10 $k \leftarrow$ SecCombine$(\vec{\alpha}')$
	11 return k

Fig. 4.11 Construction for t out of n setting.

4.4.3 t out of n setting

We now demonstrate how to use secret sharing to derive a key using only a subset of the active servers. In addition to OPRF F, the protocol uses a secret sharing scheme SSS = (SecShare, SecCombine). The user will locally run setup to acquire a vector of values, that can be stored alongside its ciphertexts at the storage server, where each entry is an OPRF output summed with a secret sharing of the user's key. This idea was used by Everspaugh *et al.* [49] in the threshold version of their OPRF system. Later, the user can rederive the file encryption key by interacting with at least t servers. If reconstruction (or file decryption) fails, the user can retry with a different subset of servers. If F is verifiable then the user can identify if a server has not responded correctly, and omit that result from the reconstruct phase.

Conceptually this scheme is quite different to the n out of n scheme in Sec. 4.4.2. The file encryption key k is generated randomly by the user of the system, rather than as a function of the user password and the OPRF keys of the servers. This does not necessarily imply it has to be sampled on the device though. Indeed, we can bootstrap the procedure by first running an n′ out of n′ scheme for $t \leq n' \leq n$ with an initially trusted subset of the servers to generate the random key k, and for most applications it would be prudent to do so.

4.5 Security Proofs

We first provide the theorems and proofs for KIND-1 security and subsequently for KIND-2, reducing the security of our construction to the PRIV-1 and PRIV-2, respectively, security of the underlying OPRF and the PRNG security of the underlying OPRF. While the t out of n case is a generalization of the n out of n, we still provide a proof for the special n out of n case as it is more instructive, and the proof is easily adaptable to the generic t out of n case.

Theorem 4.5.1. *Let* PERKS *be an* n*-out-of-*n DKA *scheme built using OPRF* F *according to Fig. 4.9 and Fig. 4.10. For any adversary* $\mathcal{A}$ *against the* KIND-1 *security of* PERKS*, there exist adversaries* $\mathcal{B}$ *and* $\mathcal{C}$ *against the* PRIV-1 *and* PRNG *security of* F *respectively, such that*

$$\mathbf{Adv}^{\text{KIND-1}}_{\mathsf{PERKS}}(\mathcal{A}, \mathrm{n}, \mathrm{n}) \leq \mathrm{n} \cdot \left(2 \cdot \mathbf{Adv}^{\text{PRIV-1}}_{\mathsf{F}}(\mathcal{B}) + \mathbf{Adv}^{\text{PRNG}}_{\mathsf{F}}(\mathcal{C}, 1) + \frac{\mathrm{q}}{|\mathcal{D}|} \right).$$

Proof Intuition. We will show a reduction from the KIND-1 to KIND′-1, where the server that may not be corrupted is fixed at the start of the game. Subsequently we will bound the KIND′-1 advantage using a sequence of game hops, starting with the $b = 0$ side where a real key is returned to the adversary and ending with the $b = 1$ side (random key). To provide a reduction to PRNG security we need to embed the PRNG challenge in one of the servers, which means that we will not be able to answer Corrupt queries for that index. To do this, we pick the server in the KIND′-1 game that may not be corrupted. The reduction from KIND-1 to KIND′-1 invokes a loss of $\frac{1}{\mathrm{n}}$. Then, for the majority of this proof we will calculate the advantage of an adversary attempting to distinguish in which game it is playing, to bound the advantage of the KIND′-1 game.

Proof. In game KIND′-1, the environment is identical to KIND-1 except that it picks a random index j out of all n servers at the start of the game and the adversary loses if it calls Corrupt on index j. We simulate KIND-1 by simply forwarding all oracle queries to KIND′-1. Given that the adversary may corrupt up to $(\mathrm{n}-1)$ servers in the course of its execution and the index j in KIND′-1 is picked uniformly at random independently of the adversary, the probability that server j will be corrupted is bounded by $1 - \frac{1}{n}$. Thus, with probability at least $\frac{1}{n}$, game KIND′-1 will not abort and the simulation succeeds, as the games are identical in this case. As a result,

$$\mathbf{Adv}^{\text{KIND-1}}_{\mathsf{PERKS}}(\mathcal{A}) \leq \mathrm{n} \cdot \mathbf{Adv}^{\text{KIND′-1}}_{\mathsf{PERKS}}(\mathcal{A}).$$

We proceed to bound $\mathbf{Adv}^{\text{KIND}'\text{-}1}_{\mathsf{PERKS}}(\mathcal{A})$ using a sequence of games G_i, and define $\epsilon_i = \mathbb{P}\left[\mathsf{G}^{G_i}_{\mathsf{PERKS}}(\mathcal{A}) = 1\right]$. Game G_0 is the $b = 0$ side, i.e. with the key returned in the Challenge query being the key computed in the protocol (if it exists) of the KIND′-1 game, and consequently $\epsilon_0 = \mathbb{P}\left[\mathsf{G}^{\text{KIND}'\text{-}1^0}_{\mathsf{PERKS}}(\mathcal{A}) = 1\right]$.

In game G_1, the environment is identical to G_0 except that for every user identity *uid*, the challenger will create two passwords: one will be used in Init, Acquire and Recover queries, and the other in Challenge and Reveal queries. Note that Reveal returns $\perp$ for any password that is not the selected password.

Intuitively, an adversary that can distinguish these games can infer information about the password or key from the req, rep values it sees from interacting with Init, Acquire and Recover, so it notices when a different password has been used in the Reveal and Challenge queries. From such an adversary we build a reduction with similar advantage against PRIV-1 of the underlying OPRF F.

The reduction $\mathcal{B}$ is detailed in Fig. 4.12. $\mathcal{B}$ plays the PRIV-1^b game and simulates the KIND′-1^0 game (G_0) or its two-password version (G_1) to $\mathcal{A}$. Let b' be the output bit of $\mathcal{A}$, i.e. its indication of which game $G_{b'}$ that $\mathcal{A}$ believes it is playing. To create the simulation, $\mathcal{B}$ selects pw_0, pw_1 for each *uid* and generates OPRF key pairs for each of the n key servers. When $\mathcal{A}$ calls Init or Acquire for some *uid*, the reduction will look up the two passwords pw_0, pw_1 associated with that user identity and call its own Challenge(uid, pw_0, pw_1) oracle and receive $(\text{req}_b, \text{req}_{1-b})$. $\mathcal{B}$ then uses secret keys for each OPRF server to produce rep_i values for req_b. Moreover, $\mathcal{B}$ computes OPRF outputs y_i for pw_0 and the user key k, to be used for Reveal and Challenge queries. Importantly, we already want to remark here that $\mathcal{B}$ will simulate G_0 if it is playing the PRIV-1^0 game (because the req, rep and k are all consistent with pw_0) and $\mathcal{B}$ will simulate G_1 if it is playing the PRIV-1^1 game (because k is derived from pw_0 and req and rep are derived from pw_1).

Acquire queries are handled similarly to Init queries, with the difference being that the OPRF output values y_i are simply stored by $\mathcal{B}$ in array Y indexed by reconstruct counter r, instead of being used to compute the key immediately. For Recover queries, the input given by the adversary is $(r, \text{C}, \vec{\text{SV}})$, and recall that in the n-out-of-n construction the $\vec{\text{SV}}$ are ignored and key reconstruction will fail if the chosen server set C is anything other than the full set of servers, i.e. $\text{C} = (\text{S}_1, \ldots, \text{S}_\text{n})$. This means the only interesting input is r, but the adversary has no control over the (deterministic) operations involved that will reconstruct the user key for password pw_0 for the *uid* corresponding with reconstruct counter r.

For queries to Reveal of the form (pw', uid), the reduction simply returns $\mathrm{K}[pw', uid]$. This will either be $\times$ if no value has been set or potentially user key k if $pw' = pw_0$ and $\mathrm{K}[pw_0, uid]$ has already been set. For Challenge(uid) queries, $\mathcal{B}$ checks if the uid has been queried before, and if not it will return k. To answer a Corrupt query, $\mathcal{B}$ needs to check if the query is allowed and **abort** otherwise, but $\mathcal{A}$ would lose anyway as this would be an illegal oracle query in both games.

As we remarked above, if $\mathcal{B}$ is playing PRIV-1^0, it will provide req, rep and k values to $\mathcal{A}$ that are consistent with each other (and consistent with password pw_0), and thus this is a perfect simulation of G_0. If $\mathcal{B}$ is playing PRIV-1^1, then pw_0 governs Challenge and Reveal queries, while pw_1 governs Init and Reconstruct queries and thus this is a perfect simulation of G_1.

It is left to argue that $\mathcal{A}$'s success in distinguishing these games carries over to an advantage for $\mathcal{B}$. Intuitively, a win for $\mathcal{A}$ implies some way of linking $(\mathrm{req}, \vec{\mathrm{rep}})$ tuples to the user keys output by the protocol in the G_0 case, or noticing the absence of such a link in the G_1 case (to see this, consider $\mathrm{n} = 1$ and a protocol where $k = pw$ and $(\mathrm{req}, \vec{\mathrm{rep}})$ information theoretically hide pw and k: an adversary has no way of distinguishing G_0 from G_1). This implies that $\mathcal{A}$ gains some information from its $(\mathrm{req}, \vec{\mathrm{rep}})$ values: if $b' = 0$ then $\mathcal{A}$ believes that its oracles are all running the same password, and thus $(\mathrm{req}_b, \vec{\mathrm{rep}})$ is linked to k, so $\mathcal{B}$ outputs 0; if $b' = 1$ then $\mathcal{A}$ thinks its oracles have been separated, and $\mathcal{B}$ outputs 1. To conclude, any advantage for $\mathcal{A}$ directly corresponds to the advantage for the reduction $\mathcal{B}$:

$$\epsilon_0 - \epsilon_1 \leq \mathbf{Adv}_{\mathsf{F}}^{\text{PRIV-1}}(\mathcal{B}).$$

In game G_2, the environment is identical to G_1 except that the Reveal and Challenge oracles return a random element of the key space. For interactions with S_j, the reduction will replace the function $\mathsf{F.Ev}(\mathrm{sk}_j, \cdot, \cdot)$ by a random function of the same domain and range, where blinded evaluation queries are simulated. Recall S_j is the randomly picked server that the adversary may not corrupt. This invokes a reduction $\mathcal{C}$ to the PRNG security of OPRF F. We show that we can use an adversary $\mathcal{A}$ that distinguishes between G_1 and G_2 to win the PRNG game with the same advantage, i.e.:

$$\epsilon_1 - \epsilon_2 \leq \mathbf{Adv}_{\mathsf{F}}^{\text{PRNG}}(\mathcal{C}, 1).$$

Let $\mathcal{C}$ play the PRNG game and simulate the KIND′-1 game (more specifically, either G_1 or G_2) to $\mathcal{A}$. The reduction is detailed in Fig. 4.13.

The reduction $\mathcal{C}$ receives a public key for its own $\mathrm{PRNG}^b(\mathcal{C}, 1)$ game, and then chooses two passwords for each user: pw_0 for Reveal and Challenge queries, and pw_1 for Init

Reduction $\mathcal{B}$ playing PRIV-1^b

00 $\mathrm{K}[\cdot] \leftarrow \perp$; $\mathrm{PW}[\cdot] \leftarrow \perp$; $\mathrm{Y}[\cdot] \leftarrow \perp$
01 For $uid \in \mathcal{UID}$:
02 $\quad pw_0, pw_1 \leftarrow_{\$} \mathcal{D}$
03 $\quad \mathrm{PW}[uid] \leftarrow pw_0, pw_1$
04 $\mathrm{CH} \leftarrow \emptyset$; $\mathrm{CO} \leftarrow \emptyset$
05 $r \leftarrow 0$
06 For $i \in [1 \,..\, \mathrm{n}]$:
07 $\quad (\mathrm{sk}_i, \mathrm{pk}_i) \leftarrow \mathsf{F.KG}$
08 $b' \leftarrow \mathcal{A}(\vec{\mathrm{pk}})$
09 Return b'

Oracle Init$(uid, \vec{\mathrm{pk}})$
10 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
11 **call** Challenge$_{\mathcal{B}}(uid, pw_0, pw_1)$
12 **receive** $(\mathrm{req}_b, \mathrm{req}_{1-b})$
13 For $i \in [1 \,..\, \mathrm{n}]$:
14 $\quad csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
15 $\quad \mathrm{rep}_i \leftarrow \mathsf{F.BlindEv}(csk_i, uid, \mathrm{req}_b)$
16 $\quad \mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
17 $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{setup}(\vec{\mathrm{y}})$
18 $\mathrm{K}[pw_0, uid] \leftarrow k$
19 Return $(\mathrm{req}_b, \vec{\mathrm{rep}}, \vec{\mathrm{SV}})$

Oracle Corrupt(i)
20 Require $i \neq j$
21 $\mathrm{CO} \xleftarrow{\cup} \{i\}$
22 Return sk_i

Oracle Acquire$(uid, \vec{\mathrm{pk}})$
23 $r \xleftarrow{+} 1$
24 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
25 **call** Challenge$_{\mathcal{B}}(uid, pw_0, pw_1)$
26 **receive** $(\mathrm{req}_b, \mathrm{req}_{1-b})$
27 For $i \in [1 \,..\, \mathrm{n}]$:
28 $\quad csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
29 $\quad \mathrm{rep}_i \leftarrow \mathsf{F.BlindEv}(csk_i, uid, \mathrm{req}_b)$
30 $\quad \mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
31 $\mathrm{Y}[r] \leftarrow (\mathrm{y}_1, \ldots, \mathrm{y}_n)$
32 Return $(\mathrm{req}_b, \vec{\mathrm{rep}}, r)$

Oracle Recover$(r, \mathrm{C}, \vec{\mathrm{SV}})$
33 $\vec{\mathrm{y}} \leftarrow \mathrm{Y}[r]$
34 $k \leftarrow \mathsf{DKA.recover}(\vec{\mathrm{y}}, \mathrm{C}, \vec{\mathrm{SV}})$
35 $\mathrm{K}[pw_0, uid] \leftarrow k$
36 Return

Oracle Challenge$_{\mathcal{A}}(uid)$
37 Require $uid \notin \mathrm{CH}$
38 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
39 $\mathrm{CH} \xleftarrow{\cup} \{uid\}$
40 $k \leftarrow \mathrm{K}[pw_0, uid]$
41 Return k

Oracle Reveal(pw', uid)
42 $k \leftarrow \mathrm{K}[pw', uid]$
43 Return k

Fig. 4.12 Reduction $\mathcal{B}$ for the proof of Theorem 4.5.1 and Theorem 4.5.2. Procedures setup and reconstruct as in Fig. 4.10 for Thm. 4.5.1 and as in Fig. 4.11 for Thm. 4.5.2.

and Reconstruct queries. It is with the uncorrupted server S_j's interactions that $\mathcal{C}$ will embed its own queries. For any query to Init or Reconstruct, the reduction $\mathcal{C}$ needs to call its own Ev oracle with pw_0 to receive the session key share y_j that will be set for future Challenge and Reveal queries, and its own BlindEv oracle with pw_1 to acquire rep_j that the adversary $\mathcal{A}$ expects to receive. Note that the user key is only set for pw_0.

To answer a Corrupt query, $\mathcal{C}$ needs to check if the query is allowed and **abort** otherwise, but $\mathcal{A}$ would lose anyway as this would be an illegal oracle query in both games. For Reveal queries on (pw', uid), the reduction simply returns $\mathrm{K}[pw', uid]$. This will either be $\times$ if no value has been set or potentially k if $pw' = pw_0$ and $\mathrm{K}[pw_0, uid]$ has already been set. For Challenge(uid) queries, $\mathcal{C}$ first checks if the uid has been queried

before, and if not it will return k. Eventually, $\mathcal{C}$ outputs to its own challenger whatever $\mathcal{A}$ outputs.

In the event that $\mathcal{C}$ is playing PRNG^0, the responses to its own queries will be the real F, and thus this perfectly simulates game G_1 for $\mathcal{A}$. If $\mathcal{C}$ is playing PRNG^1 then req and the rep values lie in the correct space but for some other randomly chosen function, and the key share for server S_j is an output of this random function, so the user key returned in Challenge is an output of a random function XORed with 'genuine' key shares, which is equivalent to choosing a random element of the key space. Thus this is a perfect simulation of G_2 for $\mathcal{A}$.

In game G_2 the Reveal and Challenge queries return a random element of the key space. Thus in G_3 we make a change to the Reveal oracle to use the key derived from pw_0 again. In all games up until this point the Reveal and Challenge oracles have been consistent with each other, but in G_3 they are not. Note that the adversary can only notice a difference between G_2 and G_3 if it queries Reveal on pw_0, as Reveal returns $\perp$ for all other passwords and the other oracles are identical. We remark that in both games Init and Reconstruct use pw_1 and Challenge simply samples a random key from the key space, so no information about pw_0 can be leaked from the oracle queries. Hence, the best any adversary can do is query the Reveal oracle for a randomly guessed password.

$$\epsilon_2 - \epsilon_3 \leq \frac{\text{q}}{|\mathcal{D}|}.$$

In game G_4 we re-merge queries such that for a given user identity *uid*, queries to Init, Reconstruct and Reveal are all associated with a single password. In a very similar manner to the hop between G_0 and G_1, this invokes a PRIV-1 term. The reduction itself is almost identical to reduction $\mathcal{B}$ in Fig. 4.12, except that line 40 is replaced by selection of a random key from the key space.

$$\epsilon_3 - \epsilon_4 \leq \mathbf{Adv}^{\text{PRIV-1}}_{\mathsf{F}}(\mathcal{B}).$$

Game G_4 is the $b = 1$ side, i.e. with the key returned in the Challenge query being a randomly chosen key, of the KIND′-1 game. Consequently $\epsilon_4 = \mathbb{P}\left[\mathsf{G}^{\text{KIND}'\text{-}1^1}_{\mathsf{PERKS}}(\mathcal{A})\right]$.

Collecting the terms results in the claimed bound, since

$$\mathbf{Adv}^{\text{KIND-1}}_{\mathsf{PERKS}}(\mathcal{A}) \leq \text{n} \cdot \mathbf{Adv}^{\text{KIND-1}'}_{\mathsf{PERKS}}(\mathcal{A}), \text{ and}$$

$$\begin{aligned}
\mathbf{Adv}^{\text{KIND}'\text{-}1}_{\mathsf{PERKS}}(\mathcal{A}) &= \left|\mathbb{P}\left[\mathsf{G}^{\text{KIND}'\text{-}1^1}_{\mathsf{PERKS}}(\mathcal{A})\right] - \mathbb{P}\left[\mathsf{G}^{\text{KIND}'\text{-}1^0}_{\mathsf{PERKS}}(\mathcal{A})\right]\right| = |\epsilon_4 - \epsilon_0| \\
&= |\epsilon_0 - \epsilon_1 + \epsilon_1 - \epsilon_2 + \epsilon_2 - \epsilon_3 + \epsilon_3 - \epsilon_4| \\
&\leq \mathbf{Adv}^{\text{PRIV-1}}_{\mathsf{F}}(\mathcal{B}) + \mathbf{Adv}^{\text{PRNG}}_{\mathsf{F}}(\mathcal{C}, 1) + \frac{\mathrm{q}}{|\mathcal{D}|} + \mathbf{Adv}^{\text{PRIV-1}}_{\mathsf{F}}(\mathcal{B}).
\end{aligned}$$

□

Reduction $\mathcal{C}$ playing $\mathrm{PRNG}^b(\mathcal{C},1)$

00 **receive** pk_j
01 $\mathrm{K}[\cdot] \leftarrow \bot$
02 $\mathrm{PW}[\cdot] \leftarrow \bot$
03 For $uid \in \mathcal{UID}$:
04 $\quad pw_0, pw_1 \leftarrow_{\$} \mathcal{D}$
05 $\quad \mathrm{PW}[uid] \leftarrow pw_0, pw_1$
06 $\mathrm{CH} \leftarrow \emptyset; \mathrm{CO} \leftarrow \emptyset$
07 For $i \in [1\,..\,\mathrm{n}] \setminus \{j\}$:
08 $\quad (\mathrm{sk}_i, \mathrm{pk}_i) \leftarrow \mathsf{F.KG}$
09 $b' \leftarrow \mathcal{A}(\vec{\mathrm{pk}})$
10 Return b'

Oracle $\mathrm{Corrupt}(i)$

11 Require $i \neq j$
12 $\mathrm{CO} \xleftarrow{\cup} \{i\}$
13 Return sk_i

Oracle $\mathrm{Init}(uid, \vec{\mathrm{pk}})$

14 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
15 **call** $\mathrm{Ev}(uid, pw_0)$
16 **receive** y
17 $\mathrm{y}_j \leftarrow \mathrm{y}$
18 $(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(uid, pw_1)$
19 **call** $\mathrm{BlindEv}(uid, \mathrm{req})$
20 **receive** rep
21 $\mathrm{rep}_j \leftarrow \mathrm{rep}$
22 For $i \in [1\,..\,\mathrm{n}] \setminus \{j\}$:
23 $\quad csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
24 $\quad \mathrm{rep}_i \leftarrow \mathsf{F.BlindEv}(csk_i, uid, \mathrm{req})$
25 $\quad \mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
26 $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{setup}(\vec{\mathrm{y}})$
27 $\mathrm{K}[pw_0, uid] \leftarrow k$
28 Return $(\mathrm{req}, \vec{\mathrm{rep}}, \vec{\mathrm{SV}})$

Oracle $\mathrm{Acquire}(uid, \vec{\mathrm{pk}})$

29 $r \xleftarrow{+} 1$
30 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
31 **call** $\mathrm{Ev}(uid, pw_0)$
32 **receive** y
33 $\mathrm{y}_i \leftarrow \mathrm{y}$
34 $(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(uid, pw_1)$
35 **call** $\mathrm{BlindEv}(uid, \mathrm{req})$
36 **receive** rep
37 $\mathrm{rep}_i \leftarrow \mathrm{rep}$
38 For $i \in [1\,..\,\mathrm{n}] \setminus \{j\}$:
39 $\quad csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
40 $\quad \mathrm{rep}_i \leftarrow \mathsf{F.BlindEv}(csk_i, uid, \mathrm{req})$
41 $\quad \mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
42 $\mathrm{Y}[r] \leftarrow (\mathrm{y}_1, \ldots, \mathrm{y}_\mathrm{n})$
43 Return $(\mathrm{req}, \vec{\mathrm{rep}}, r)$

Oracle $\mathrm{Recover}(r, \mathrm{C}, \vec{\mathrm{SV}})$

44 $\vec{\mathrm{y}} \leftarrow \mathrm{Y}[r]$
45 $k \leftarrow \mathsf{DKA.recover}(\vec{\mathrm{y}}, \mathrm{C}, \vec{\mathrm{SV}})$
46 $\mathrm{K}[pw_0, uid] \leftarrow k$
47 Return

Oracle $\mathrm{Reveal}(pw', uid)$

48 $k \leftarrow \mathrm{K}[pw', uid]$
49 Return k

Oracle $\mathrm{Challenge}_{\mathcal{A}}(uid)$

50 Require $uid \notin \mathrm{CH}$
51 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
52 $\mathrm{CH} \xleftarrow{\cup} \{uid\}$
53 $k \leftarrow \mathrm{K}[pw_0, uid]$
54 Return k

Fig. 4.13 Reduction $\mathcal{C}$ for the proof of Theorem 4.5.1. Procedures setup and reconstruct as in Fig. 4.10.

Theorem 4.5.2. *Let* PERKS *be an* t*-out-of-*n DKA *scheme built using OPRF* F *according to Fig. 4.9 and Fig. 4.11 for* t *such that* $1 \leq t \leq n$. *For any adversary* $\mathcal{A}$ *against the* KIND-1 *security of* PERKS*, there exist adversaries* $\mathcal{B}$ *and* $\mathcal{C}$ *against the* PRIV-1 *and* PRNG *security of* F *respectively, such that*

$$\mathbf{Adv}^{\text{KIND-1}}_{\mathsf{PERKS}}(\mathcal{A}, t, n) \leq \binom{n}{t-1} \cdot \left(2 \cdot \mathbf{Adv}^{\text{PRIV-1}}_{\mathsf{F}}(\mathcal{B}) + \mathbf{Adv}^{\text{PRNG}}_{\mathsf{F}}(\mathcal{C}, n-t+1) + \frac{q}{|\mathcal{D}|}\right).$$

Proof Sketch. We remark Theorem 4.5.1 is the special case $t = n$ of this theorem and the game hops are very similar to that proof. For brevity we only provide the modifications to the proof here rather than duplicating the proof in its entirety. We also believe that only highlighting the steps where the proof needs to be generalized increases clarity.

For the first step, the success probability of the simulation now depends on t, as the reduction needs to select $n-t+1$ uncorrupted servers. We denote this set of indices of uncorrupted servers with J. A lower bound for the success probability is provided by the number of ways to select $(t-1)$ servers out of $(t-1)$ servers divided by the number of ways to select $(t-1)$ servers out of n servers:

$$\frac{1}{\binom{n}{t-1}}.$$

Note that for $t = n$ we obtain the lower bound $\frac{1}{n}$ from Theorem 4.5.1.

The reduction $\mathcal{B}$ in Fig. 4.12 from the indistinguishability between G_0 and G_1 to the PRIV-1 game is the same as in Theorem 4.5.1 with the trivial modification that it uses the t out of n setup and reconstruct procedures from Fig. 4.11 instead of the procedures from Fig. 4.10. We apply the same modification to the reduction $\mathcal{C}$ from the indistinguishability between G_1 and G_2 to the PRNG game. Moreover, the special case $i = j$, where j is the index of the uncorrupted server in reduction $\mathcal{C}$ is now generalized to $i \in J$, where J is the set of indices of uncorrupted servers. For completeness we provide the updated reduction in Fig. 4.14, but intuitively nothing novel happens in the reduction.

We need to argue replacing XOR in the setup and reconstruct procedures with a key sharing scheme also simulates G_2, i.e. the selected key is a random element from the key space. It is clear the adversary can have at most $(t-1)$ 'genuine' key shares because $(n-t+1)$ servers return a random element. By the security of the secret sharing scheme, with $(t-1)$ key shares, any $k \in \mathcal{K}$ can still be reconstructed. Thus, if key share $s_t \in \mathcal{K}$ is

a random element of the key space, then so is the reconstructed key. It is clear this holds as s_t is the XOR of $\vec{SV}_t$ and y_t, where y_t is a random element of $\mathcal{K}$.

There is no modification to the hop from G_2 to G_3. The final hop from G_3 to G_4 is again the same as in in Theorem 4.5.1 with the trivial modification that it uses the t out of n setup and reconstruct procedures from Fig. 4.11. Collecting the terms yields the claimed result.

Reduction $\mathcal{C}$ playing $\mathrm{PRNG}^b(\mathcal{C}, \mathrm{n}-\mathrm{t}+1)$

00 **receive** $\vec{\mathrm{pk}}'$
01 $\mathrm{K}[\cdot] \leftarrow \perp$; $\mathrm{PW}[\cdot] \leftarrow \perp$
02 For $uid \in \mathcal{UID}$:
03 $pw_0, pw_1 \leftarrow_{\$} \mathcal{D}$
04 $\mathrm{PW}[uid] \leftarrow pw_0, pw_1$
05 $\mathrm{CH} \leftarrow \emptyset$; $\mathrm{CO} \leftarrow \emptyset$
06 For $i \in J$:
07 $\mathrm{pk}_i \leftarrow \mathrm{pk}'_{\sigma(i)}$
08 For $i \in [1\,..\,\mathrm{n}] \setminus J$:
09 $(\mathrm{sk}_i, \mathrm{pk}_i) \leftarrow \mathsf{F.KG}$
10 $b' \leftarrow \mathcal{A}(\vec{\mathrm{pk}})$
11 Return b'

Oracle $\mathrm{Init}(uid, \vec{\mathrm{pk}})$
12 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
13 $(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(uid, pw_1)$
14 For $i \in J$:
15 **call** $\mathrm{Ev}_{\sigma(i)}(uid, pw_0)$
16 **receive** y_i
17 **call** $\mathrm{BlindEv}_{\sigma(i)}(uid, \mathrm{req})$
18 **receive** rep_i
19 For $i \in [1\,..\,\mathrm{n}] \setminus J$:
20 $csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
21 $\mathrm{rep}_i \leftarrow \mathsf{F.BlindEv}(csk_i, uid, \mathrm{req})$
22 $\mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
23 $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{setup}(\vec{\mathrm{y}})$
24 $\mathrm{K}[pw_0, uid] \leftarrow k$
25 Return $(\mathrm{req}, \vec{\mathrm{rep}}, \vec{\mathrm{SV}})$

Oracle $\mathrm{Reveal}(pw', uid)$
26 $k \leftarrow \mathrm{K}[pw', uid]$
27 Return k

Oracle $\mathrm{Acquire}(uid, \vec{\mathrm{pk}})$
28 $r \xleftarrow{+} 1$
29 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
30 $(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(uid, pw_1)$
31 For $i \in J$:
32 **call** $\mathrm{Ev}_{\sigma(i)}(uid, pw_0)$
33 **receive** y_i
34 **call** $\mathrm{BlindEv}_{\sigma(i)}(uid, \mathrm{req})$
35 **receive** rep_i
36 For $i \in [1\,..\,\mathrm{n}] \setminus J$:
37 $csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
38 $\mathrm{rep}_i \leftarrow \mathsf{F.BlindEv}(csk_i, uid, \mathrm{req})$
39 $\mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
40 $\mathrm{Y}[r] \leftarrow (\mathrm{y}_1, \ldots, \mathrm{y}_\mathrm{n})$
41 Return $(\mathrm{req}, \vec{\mathrm{rep}}, r)$

Oracle $\mathrm{Recover}(r, \mathrm{C}, \vec{\mathrm{SV}})$
42 $\vec{\mathrm{y}} \leftarrow \mathrm{Y}[r]$
43 $k \leftarrow \mathsf{DKA.recover}(\vec{\mathrm{y}}, \mathrm{C}, \vec{\mathrm{SV}})$
44 $\mathrm{K}[pw_0, uid] \leftarrow k$
45 Return

Oracle $\mathrm{Challenge}_{\mathcal{A}}(uid)$
46 Require $uid \notin \mathrm{CH}$
47 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
48 $\mathrm{CH} \xleftarrow{\cup} \{uid\}$
49 $k \leftarrow \mathrm{K}[pw_0, uid]$
50 Return k

Oracle $\mathrm{Corrupt}(i)$
51 Require $i \notin J$
52 $\mathrm{CO} \xleftarrow{\cup} \{i\}$
53 Return sk_i

Fig. 4.14 Reduction $\mathcal{C}$ for the proof of Theorem 4.5.2. Procedures setup and reconstruct as in Fig. 4.11. J is a set of t indices that may not be corrupted. σ is a bijection of the uncorrupted indices in the KIND game to the indices in the underlying PRNG game.

We now provide the theorems and proofs for KIND-2 security. For completeness we provide the modified reductions but as the modifications are trivial the descriptions will be brief. The theorems bound the advantage by $\mathbf{Adv}^{\text{PRIV-2}}_{\mathsf{F}}(\mathcal{B})$ (instead of PRIV-1) and in the proof we only need to adapt the reductions to use the F.finalize procedure and the Finalize oracle in the PRIV-2 game to compute the OPRF output value (instead of using the F.ev procedure and the Ev oracle), since the response may now be maliciously formed.

Theorem 4.5.3. *Let* PERKS *be an* n*-out-of-*n DKA *scheme built using OPRF* F *according to Fig. 4.9 and Fig. 4.10. For any adversary* $\mathcal{A}$ *against the* KIND-2 *security of* PERKS*, there exist adversaries* $\mathcal{B}$ *and* $\mathcal{C}$ *against the* PRIV-2 *and* PRNG *security of* F *respectively, such that*

$$\mathbf{Adv}^{\text{KIND-2}}_{\mathsf{PERKS}}(\mathcal{A}, \mathrm{n}, \mathrm{n}) \leq \mathrm{n} \cdot \left(2 \cdot \mathbf{Adv}^{\text{PRIV-2}}_{\mathsf{F}}(\mathcal{B}) + \mathbf{Adv}^{\text{PRNG}}_{\mathsf{F}}(\mathcal{C}) + \frac{\mathrm{q}}{|\mathcal{D}|}\right).$$

Proof Sketch. The proof goes analogously to the proof of Theorem 4.5.1. We need to adapt the reductions as they cannot assume functionality and simply call the F.ev procedure or the Ev oracle, since the rep values may now be maliciously formed. Therefore, the reductions now use the F.finalize procedure and the Finalize oracle in the PRIV-2 game to compute the OPRF output value y (or receive $\perp$). The modifications are trivial but for completeness we provide the updated reductions here. The reduction $\mathcal{B}$ is detailed in Fig. 4.15 and reduction $\mathcal{C}$ is detailed in Fig. 4.16.

Theorem 4.5.4. *Let* PERKS *be an* t*-out-of-*n DKA *scheme built using OPRF* F *according to Fig. 4.9 and Fig. 4.11 for* t *such that* $1 \leq \mathrm{t} \leq \mathrm{n}$*. For any adversary* $\mathcal{A}$ *against the* KIND-2 *security of* PERKS*, there exist adversaries* $\mathcal{B}$ *and* $\mathcal{C}$ *against the* PRIV-2 *and* PRNG *security of* F *respectively, such that*

$$\mathbf{Adv}^{\text{KIND-2}}_{\mathsf{PERKS}}(\mathcal{A}, \mathrm{t}, \mathrm{n}) \leq \binom{\mathrm{n}}{\mathrm{t}-1} \cdot \left(2 \cdot \mathbf{Adv}^{\text{PRIV-2}}_{\mathsf{F}}(\mathcal{B}) + \mathbf{Adv}^{\text{PRNG}}_{\mathsf{F}}(\mathcal{C}, \mathrm{n}-\mathrm{t}+1) + \frac{\mathrm{q}}{|\mathcal{D}|}\right).$$

Proof Sketch. This proof effectively applies both the adaptations made in Theorem 4.5.2 and Theorem 4.5.3. The reduction $\mathcal{C}$ is detailed in Fig. 4.17. Reduction $\mathcal{B}$ is the same as in Theorem 4.5.3 with the trivial modification that it uses the t out of n setup and reconstruct procedures from Fig. 4.11 instead of the procedures from Fig. 4.10.

Reduction $\mathcal{B}$ playing PRIV-2^b

00 $\mathrm{K}[\cdot] \leftarrow \bot$; $\mathrm{PW}[\cdot] \leftarrow \bot$; $\mathrm{Y}[\cdot] \leftarrow \bot$
01 For $uid \in \mathcal{UID}$:
02 $\quad pw_0, pw_1 \leftarrow_{\$} \mathcal{D}$
03 $\quad \mathrm{PW}[uid] \leftarrow pw_0, pw_1$
04 $\mathrm{CH} \leftarrow \emptyset$; $\mathrm{CO} \leftarrow \emptyset$
05 $r \leftarrow 0$
06 For $i \in [1\,..\,\mathrm{n}]$:
07 $\quad (\mathrm{sk}_i, \mathrm{pk}_i) \leftarrow \mathsf{F.KG}$
08 $b' \leftarrow \mathcal{A}(\vec{\mathrm{pk}})$
09 Return b'

Oracle $\mathrm{Init}(uid, \vec{\mathrm{pk}})$
10 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
11 **call** $\mathrm{Challenge}_{\mathcal{B}}(uid, pw_0, pw_1)$
12 **receive** $(j, \mathrm{req}_b, \mathrm{req}_{1-b})$
13 For $i \in [1\,..\,\mathrm{n}]$:
14 $\quad$ Await $\mathrm{rep}_b^i \leftarrow \mathcal{A}.\mathsf{server.op}_i(uid, \mathrm{req}_b)$
15 $\quad$ Await $\mathrm{rep}_{1-b}^i \leftarrow \mathcal{A}.\mathsf{server.op}_i(uid, \mathrm{req}_{1-b})$
16 $\quad cpk_i \leftarrow \mathsf{DKA.Cli.PKG}(\mathrm{pk}_i, uid)$
17 $\quad$ **call** $\mathrm{Finalize}_{\mathcal{B}}(j, cpk_i, \mathrm{rep}_b^i, \mathrm{rep}_{1-b}^i)$
18 $\quad$ **receive** $(\mathrm{y}_0^i, \mathrm{y}_1^i)$
19 $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{setup}(\vec{\mathrm{y}}_0)$
20 $\mathrm{K}[pw_0, uid] \leftarrow k$
21 Return $\vec{\mathrm{SV}}$

Oracle $\mathrm{Challenge}_{\mathcal{A}}(uid)$
22 Require $uid \notin \mathrm{CH}$
23 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
24 $\mathrm{CH} \xleftarrow{\cup} \{uid\}$
25 $k \leftarrow \mathrm{K}[pw_0, uid]$
26 Return k

Oracle $\mathrm{Acquire}(uid, \vec{\mathrm{pk}})$
27 $r \xleftarrow{+} 1$
28 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
29 **call** $\mathrm{Challenge}_{\mathcal{B}}(uid, pw_0, pw_1)$
30 **receive** $(j, \mathrm{req}_b, \mathrm{req}_{1-b})$
31 For $i \in [1\,..\,\mathrm{n}]$:
32 $\quad$ Await $\mathrm{rep}_b^i \leftarrow \mathcal{A}.\mathsf{server.op}_i(uid, \mathrm{req}_b)$
33 $\quad$ Await $\mathrm{rep}_{1-b}^i \leftarrow \mathcal{A}.\mathsf{server.op}_i(uid, \mathrm{req}_{1-b})$
34 $\quad cpk_i \leftarrow \mathsf{DKA.Cli.PKG}(\mathrm{pk}_i, uid)$
35 $\quad$ **call** $\mathrm{Finalize}_{\mathcal{B}}(j, cpk_i, \mathrm{rep}_b^i, \mathrm{rep}_{1-b}^i)$
36 $\quad$ **receive** $(\mathrm{y}_0^i, \mathrm{y}_1^i)$
37 $\mathrm{Y}[r] \leftarrow (\mathrm{y}_0^1, \ldots, \mathrm{y}_0^{\mathrm{n}})$
38 Return r

Oracle $\mathrm{Recover}(r, \mathrm{C}, \vec{\mathrm{SV}})$
39 $\vec{\mathrm{y}} \leftarrow \mathrm{Y}[r]$
40 $k \leftarrow \mathsf{DKA.recover}(\vec{\mathrm{y}}, \mathrm{C}, \vec{\mathrm{SV}})$
41 $\mathrm{K}[pw_0, uid] \leftarrow k$
42 Return

Oracle $\mathrm{BlindEv}(i, uid, \mathrm{req})$
43 $\mathrm{rep}_i \leftarrow \mathsf{DKA.server.op}(\mathrm{sk}_i, uid, \mathrm{req})$
44 Return rep_i

Oracle $\mathrm{Reveal}(pw', uid)$
45 $k \leftarrow \mathrm{K}[pw', uid]$
46 Return k

Oracle $\mathrm{Corrupt}(i)$
47 Require $i \neq j$
48 $\mathrm{CO} \xleftarrow{\cup} \{i\}$
49 Return sk_i

Fig. 4.15 Reduction $\mathcal{B}$ for the proof of Theorem 4.5.3 and Theorem 4.5.4. Procedures setup and reconstruct as in Fig. 4.10 for Thm. 4.5.3 and as in Fig. 4.11 for Thm. 4.5.4.

Reduction $\mathcal{C}$ playing $\mathrm{PRNG}^b(\mathcal{C},1)$

00 **receive** pk_j
01 $\mathrm{K}[\cdot] \leftarrow \bot$; $\mathrm{PW}[\cdot] \leftarrow \bot$
02 For $uid \in \mathcal{UID}$:
03 $pw_0, pw_1 \leftarrow_\$ \mathcal{D}$
04 $\mathrm{PW}[uid] \leftarrow pw_0, pw_1$
05 $\mathrm{CH} \leftarrow \emptyset$; $\mathrm{CO} \leftarrow \emptyset$
06 For $i \in [1\,..\,\mathrm{n}] \setminus \{j\}$:
07 $(\mathrm{sk}_i, \mathrm{pk}_i) \leftarrow \mathsf{F.KG}$
08 $b' \leftarrow \mathcal{A}(\vec{\mathrm{pk}})$
09 Return b'

Oracle $\mathrm{Init}(uid, \vec{\mathrm{pk}})$

10 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
11 $(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(uid, pw_1)$
12 For $i \in [1\,..\,\mathrm{n}]$:
13 Await $\mathrm{rep}_i \leftarrow \mathcal{A}.\mathsf{server.op}_i(uid, \mathrm{req})$
14 If $i \neq j$:
15 $csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
16 $\mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
17 Else:
18 **call** $\mathrm{Ev}(uid, pw_0)$
19 **receive** y_i
20 $cpk_i \leftarrow \mathsf{DKA.Cli.PKG}(\mathrm{pk}_i, uid)$
21 $\mathrm{y}'_i \leftarrow \mathsf{F.Finalize}(cpk_i, \mathrm{rep}_i, \mathrm{st})$
22 If $\mathrm{y}'_i = \bot$:
23 $\mathrm{y}_i \leftarrow \bot$
24 $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{setup}(\vec{\mathrm{y}})$
25 $\mathrm{K}[pw_0, uid] \leftarrow k$
26 Return $\vec{\mathrm{SV}}$

Oracle $\mathrm{Challenge}_{\mathcal{A}}(uid)$

27 Require $uid \notin \mathrm{CH}$
28 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
29 $\mathrm{CH} \xleftarrow{\cup} \{uid\}$
30 $k \leftarrow \mathrm{K}[pw_0, uid]$
31 Return k

Oracle $\mathrm{Reveal}(pw', uid)$

32 $k \leftarrow \mathrm{K}[pw', uid]$
33 Return k

Oracle $\mathrm{Acquire}(uid, \vec{\mathrm{pk}})$

34 $r \xleftarrow{+} 1$
35 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
36 $(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(uid, pw_1)$
37 For $i \in [1\,..\,\mathrm{n}]$:
38 Await $\mathrm{rep}_i \leftarrow \mathcal{A}.\mathsf{server.op}_i(uid, \mathrm{req})$
39 If $i \neq j$:
40 $csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
41 $\mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
42 Else:
43 **call** $\mathrm{Ev}(uid, pw_0)$
44 **receive** y_i
45 $cpk_i \leftarrow \mathsf{DKA.Cli.PKG}(\mathrm{pk}_i, uid)$
46 $\mathrm{y}'_i \leftarrow \mathsf{F.Finalize}(cpk_i, \mathrm{rep}_i, \mathrm{st})$
47 If $\mathrm{y}'_i = \bot$:
48 $\mathrm{y}_i \leftarrow \bot$
49 $\mathrm{Y}[r] \leftarrow (\mathrm{y}_1, \ldots, \mathrm{y}_\mathrm{n})$
50 Return $(\mathrm{req}, \vec{\mathrm{rep}}, r)$

Oracle $\mathrm{Recover}(r, \mathrm{C}, \vec{\mathrm{SV}})$

51 $\vec{\mathrm{y}} \leftarrow \mathrm{Y}[r]$
52 $k \leftarrow \mathsf{DKA.recover}(\vec{\mathrm{y}}, \mathrm{C}, \vec{\mathrm{SV}})$
53 $\mathrm{K}[pw_0, uid] \leftarrow k$
54 Return

Oracle $\mathrm{BlindEv}(i, uid, \mathrm{req})$

55 If $i \neq j$:
56 $\mathrm{rep} \leftarrow \mathsf{DKA.server.op}(\mathrm{sk}_i, uid, \mathrm{req})$
57 Else:
58 **call** $\mathrm{BlindEv}(uid, \mathrm{req})$
59 **receive** rep
60 Return rep

Oracle $\mathrm{Corrupt}(i)$

61 Require $i \neq j$
62 $\mathrm{CO} \xleftarrow{\cup} \{i\}$
63 Return sk_i

Fig. 4.16 Reduction $\mathcal{C}$ for the proof of Theorem 4.5.3. Procedures setup and reconstruct as in Fig. 4.10.

Reduction $\mathcal{C}$ playing $\mathrm{PRNG}^b(\mathcal{C}, \mathrm{n}-\mathrm{t}+1)$

00 **receive** $\vec{\mathrm{pk}}'$
01 $\mathrm{K}[\cdot] \leftarrow \perp$; $\mathrm{PW}[\cdot] \leftarrow \perp$
02 For $uid \in \mathcal{UID}$:
03 $pw_0, pw_1 \leftarrow_{\$} \mathcal{D}$
04 $\mathrm{PW}[uid] \leftarrow pw_0, pw_1$
05 $\mathrm{CH} \leftarrow \emptyset$; $\mathrm{CO} \leftarrow \emptyset$
06 For $i \in J$:
07 $\mathrm{pk}_i \leftarrow \mathrm{pk}'_{\sigma(i)}$
08 For $i \in [1\,..\,\mathrm{n}] \setminus J$:
09 $(\mathrm{sk}_i, \mathrm{pk}_i) \leftarrow \mathsf{F.KG}$
10 $b' \leftarrow \mathcal{A}(\vec{\mathrm{pk}})$
11 Return b'

Oracle $\mathrm{Init}(uid, \vec{\mathrm{pk}})$

12 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
13 $(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(uid, pw_1)$
14 For $i \in [1\,..\,\mathrm{n}]$:
15 Await $\mathrm{rep}_i \leftarrow \mathcal{A}.\mathsf{server.op}_i(uid, \mathrm{req})$
16 If $i \in J$:
17 **call** $\mathrm{Ev}_{\sigma(i)}(uid, pw_0)$
18 **receive** y_i
19 If $i \notin J$:
20 $csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
21 $\mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
22 $cpk_i \leftarrow \mathsf{DKA.Cli.PKG}(\mathrm{pk}_i, uid)$
23 $\mathrm{y}'_i \leftarrow \mathsf{F.Finalize}(cpk_i, \mathrm{rep}_i, \mathrm{st})$
24 If $\mathrm{y}'_i = \perp$:
25 $\mathrm{y}_i \leftarrow \perp$
26 $(k, \vec{\mathrm{SV}}) \leftarrow \mathrm{setup}(\vec{\mathrm{y}})$
27 $\mathrm{K}[pw_0, uid] \leftarrow k$
28 Return $\vec{\mathrm{SV}}$

Oracle $\mathrm{Challenge}_{\mathcal{A}}(uid)$

29 Require $uid \notin \mathrm{CH}$
30 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
31 $\mathrm{CH} \xleftarrow{\cup} \{uid\}$
32 $k \leftarrow \mathrm{K}[pw_0, uid]$
33 Return k

Oracle $\mathrm{Acquire}(uid, \vec{\mathrm{pk}})$

34 $r \xleftarrow{+} 1$
35 $pw_0, pw_1 \leftarrow \mathrm{PW}[uid]$
36 $(\mathrm{req}, \mathrm{st}) \leftarrow \mathsf{F.Req}(uid, pw_1)$
37 For $i \in [1\,..\,\mathrm{n}]$:
38 Await $\mathrm{rep}_i \leftarrow \mathcal{A}.\mathsf{server.op}_i(uid, \mathrm{req})$
39 If $i \in J$:
40 **call** $\mathrm{Ev}_{\sigma(i)}(uid, pw_0)$
41 **receive** y_i
42 If $i \notin J$:
43 $csk_i \leftarrow \mathsf{DKA.Cli.SKG}(\mathrm{sk}_i, uid)$
44 $\mathrm{y}_i \leftarrow \mathsf{F.Ev}(csk_i, uid, pw_0)$
45 $cpk_i \leftarrow \mathsf{DKA.Cli.PKG}(\mathrm{pk}_i, uid)$
46 $\mathrm{y}'_i \leftarrow \mathsf{F.Finalize}(cpk_i, \mathrm{rep}_i, \mathrm{st})$
47 If $\mathrm{y}'_i = \perp$:
48 $\mathrm{y}_i \leftarrow \perp$
49 $\mathrm{Y}[r] \leftarrow (\mathrm{y}_1, \ldots, \mathrm{y}_\mathrm{n})$
50 Return $(\mathrm{req}, \vec{\mathrm{rep}}, r)$

Oracle $\mathrm{Recover}(r, \mathrm{C}, \vec{\mathrm{SV}})$

51 $\vec{\mathrm{y}} \leftarrow \mathrm{Y}[r]$
52 $k \leftarrow \mathsf{DKA.recover}(\vec{\mathrm{y}}, \mathrm{C}, \vec{\mathrm{SV}})$
53 $\mathrm{K}[pw_0, uid] \leftarrow k$
54 Return

Oracle $\mathrm{BlindEv}(i, uid, \mathrm{req})$

55 If $i \in J$:
56 **call** $\mathrm{BlindEv}_{\sigma(i)}(uid, \mathrm{req})$
57 **receive** rep
58 Else:
59 $\mathrm{rep} \leftarrow \mathsf{DKA.server.op}(\mathrm{sk}_i, uid, \mathrm{req})$
60 Return rep

Oracle $\mathrm{Reveal}(pw', uid)$

61 $k \leftarrow \mathrm{K}[pw', uid]$
62 Return k

Oracle $\mathrm{Corrupt}(i)$

63 Require $i \notin J$
64 $\mathrm{CO} \xleftarrow{\cup} \{i\}$
65 Return sk_i

Fig. 4.17 Reduction $\mathcal{C}$ for the proof of Theorem 4.5.4. Procedures setup and reconstruct as in Fig. 4.11. J is a set of t indices that may not be corrupted. σ is a bijection of the uncorrupted indices in the KIND game to the indices in the underlying PRNG game.

4.6 Use of Existing OPRFs in PERKS

As we have mentioned in Sec. 4.4, the DH-based VOPRFs in Fig. 4.2 allow the server to store one master key and compute private and public keys for users on the fly using *uid*: this operation is specified in Fig. 4.9. Remember that for non-verifiable OPRFs there is no public key and thus on-the-fly computation of per-user key material just needs to run a key derivation function from the server's (single) master key sk and *uid* to the same space as sk.

For non-DH VOPRFs, the DH group trick is not directly applicable, so either a similar trick using the structure of the public and secret keys needs to be found, or the server needs to store per-user key material. We regard finding such tricks in post-quantum VOPRFs as future work. The CSIDH-based scheme of Boneh *et al.* [25] is not defined as a VOPRF, however this would appear to be a good candidate for a VOPRF that could fit with our DH trick.

For key rotation, the Pythia OPRF has no 'outer hash' (that destroys algebraic structure) and so is eligible for simple key rotation. Note that the aforementioned HashDH scheme can provide key rotation but only if the user stores inner hash values, but this is undesirable in our setting and modelling security for this case is not trivial.

This invokes a tradeoff: the Pythia OPRF provides key rotation at a computational cost (due to the pairing operation), while 2HashDH and 3HashSDHI are fast but without key rotation. As a result, the system designer needs to judge if the 'user initiated' key rotation methods in Sec. 4.7 are viable for the system's users, and if so 2HashDH or 3HashSDHI can be used.

Note that each of the three OPRFs in Fig. 4.2 are proven secure in different models, and only the 3HashSDHI scheme has been proven secure in our OPRF security games. Thus it remains to formally prove that the other two schemes do in fact meet PRIV-x and PRNG security, or by showing that the proven security properties of the other schemes—VOPRF UC functionality for 2HashDH, and one-more unpredictability and one-more PRF for Pythia—are at least as strong as PRIV-x and PRNG.

4.7 Using PERKS as a Storage System

We now explain why our approach is well-suited to derivation of a backup key for outsourced storage systems, and particularly for instant messaging. Then, we describe how our construction can be used to build a feature rich file system for cloud storage,

incorporating recent work analysing security of symmetric encryption schemes where a user encrypts 'to themself', deduplication, and efficient key rotation.

Instant messaging apps are generally free to download and use, and users are often unwilling to pay for additional features. This leaves very little room for manoeuvre when designing a secure backup service: users must use an internal solution like WhatsApp's (see Sec. 4.1.3), where the protocol is potentially strong but not open source, so users have to place their trust in the provider. A service such as the one we propose needs to be extremely efficient in terms of bandwidth and storage to possibly be offered as a free service: this is why we aim to only use the most efficient OPRFs, and enforce minimal user and server storage. In particular, we envision OPRF services with multiple other roles in addition to PERKS, hence our system does not require the OPRF servers to be given a particular (share of a) key, as is done in many prior works [60, 62, 15].

The constructions defined in Sec. 4.4 allow a user to derive a single symmetric key from a password. It remains to select a symmetric encryption primitive for encryption, a decision that is informed by the desired functionality and security properties.

Note that in the case of long-term encrypted backup, if a user's device is compromised and they wish to change their encryption key, they may still wish to recover messages stored under the old key (i.e. even if they believe that an adversary is already in possession of those messages). From this perspective, the user may wish to recover their messages after they have already chosen a new password for use with new messages, creating an overlap in the epochs of the system: this is a departure from the regular theoretical approach to key rotation via updatable encryption and we discuss this further below.

Encrypt-to-Self. In Chapter 3 we showed that integrity protection can still be obtained in the event of user key corruption: if the user stores short file (ciphertext) hashes then even if the user knows that their key is corrupted they can check ciphertext integrity when downloading files and discard any where the hash does not match a local entry. We demonstrated a method to compute these hashes during encryption, to avoid making two passes over plaintext data.

Deduplication. If the user expects to upload some files many times, for example by backing up an entire disk periodically, and wants to avoid storing multiple copies of files then they can employ deduplication techniques such as convergent encryption [93, 17]. File key derivation for a file F could be for example $k_F \leftarrow \mathsf{H}(k||F)$ for some cryptographic hash function H.

Key Rotation. If the user wishes to rotate their file encryption key in PERKS then there are three possibilities:

1. Use an OPRF service that has automated key rotation, e.g. by using the Pythia OPRF [49]. Note that for the n out of n construction, just one OPRF server updating its sk_i value results in a change in file encryption key. If this is used, then the server will provide 'tokens' that work similarly to updatable encryption (UE) update tokens: unblinded values provided under the old key can be efficiently modified to unblinded values under the new key, without the need to call the OPRF service under all the old inputs.
2. Use a different password. This will result in new OPRF output values for all OPRF servers. (Note that another credential modification technique is possible via an OPRF service that supports tweaks, i.e. different *uid* input values for the same user: this will result in different OPRF output values for the OPRF servers offering this.)
3. (t out of n construction only) Choose a new key k, essentially running setup again. This results in a new key share vector $\vec{\alpha}$ but the y_i values are unchanged so the user needs to publish a new public vector $\vec{\mathrm{SV}}$.

In all of these cases, the user can avoid downloading, decrypting, reencrypting and reuploading all of their files every time they update their file encryption key by utilizing updatable encryption [26, 72, 29], where the user can send a short update token to the ciphertext storage server (CSP) with which the underlying key for the ciphertexts can be rotated efficiently, without leaking information to the CSP. However the challenge is providing availability of key material in consecutive epochs. In the efficient (ciphertext-independent) UE schemes just mentioned, the update token calculation requires knowledge of an old key and a new key at the beginning of the new epoch. For user-initiated actions (items 2 and 3) this is trivial: the user runs the protocol to get their old key, then runs the protocol again using their new inputs, calculates the token, sends that to the file storage server and then deletes both keys locally. In the OPRF server key rotation setting (item 1) care is required: if a new epoch begins while the user does not have a local copy of the file encryption key available then the user would be locked out of access to their ciphertexts. To solve this issue the OPRF services could make a transition period available to users, where access is given to the OPRF functionality for the old and the new OPRF keys.

4.8 Conclusion and Reflection

With the exception of the PRNG definition that is inherited from the work of Tyagi *et al.* [96] and adapted to our multi-server setting, the security definitions presented in this thesis are game-based rather than simulation-based. Many papers in the literature present OPRF security in the universal composability framework [35] which reflects the fact that OPRFs are regularly composed with other primitives and are used in highly concurrent scenarios. We avoided using the UC model for our formalization of KIND-x due to the inherent requirement in the UC framework for the parties to agree on a session identifier sid before an instance of the protocol begins. It is of course trivial to agree on sid either via an additional round of communication or using counters stored at each party, however both of these options are extremely undesirable in the scenario that we consider. We consider it valuable future work to define a UC functionality that captures key acquisition with the same properties as our KIND-x games, with the additional benefit of composability.

At USENIX '15, Everspaugh *et al.* [49] (ECSJR) presented the first definition of partially-oblivious OPRFs (POPRFs) and a candidate construction using pairings (more details in Sec. 4.2.3). Their motivating setting was a user engaging in password-based authentication with some web server, where instead of the web server storing password hashes, it would interact with 'a Pythia service' that hardened the user's password using a POPRF and a random per-user value as the OPRF public input t. ECSJR used one-more unpredictability and one-more pseudorandomness as properties of the POPRF but do not consider any formal security properties for the use cases that motivated their paper, whereas in our case we needed to create KIND-x as a security definition that captures the indistinguishability of the keys used by the user in the eventual application. An interesting future research topic would be to categorize the security definitions for (P)OPRFs, to ascertain the 'correct' expectations of security. To provide a brief summary of the difficulties in comparing notions, Tyagi *et al.* explicitly referred to the ECSJR one-more pseudorandomness property as "non-standard" and advertise their own notions as avoiding the need for the ECSJR approach, while ECSJR provide a very brief sketch (in the ePrint version [50] only) of the difficulty in proving their own POPRF UC secure (there is no outer hash because they seek key rotation, and this makes a UC proof nigh on impossible because the simulator can never see the necessary random oracle queries), followed by an unproven claim that adding an extra outer hash gives a UC-secure VOPRF.

Finally, we remark that our approach defines an OPRF interaction as being two messages, namely a blinded representation of the password from the user to the server and then an application of the OPRF secret key to this blinded value in response. It is

known how to construct (V)OPRFs with post-quantum security in this (round-optimal) manner [3] (see Sec. 4.6 for more details on existing P- and V-OPRF constructions), however it is still unclear if efficient constructions are feasible, and such constructions may not follow the two-message syntax that we use. In the event that relatively efficient verifiable OPRFs with post-quantum security can be constructed using some other syntax then firstly this would be a significant breakthrough, and further it would be necessary to adapt our formalism.

Chapter 5

Conclusion

In this thesis we have extensively discussed the security of instant messaging, an ubiquitous communication tool in the modern world. ACD [5] observed that research on secure messaging protocols routinely only considers settings with a guaranteed in-order delivery of messages, while most real-world protocols like Signal are actually designed for out-of-order delivery. We reassessed the model and construction of ACD and argue that there is not only a mismatch between theory and practice, but there is also a mismatch between what users intuitively expect and what cryptographers deliver. The intuitive notion of forward secrecy is not provided: One would expect past *sent* messages to remain confidential after an exposure, however academic literature has only focused on confidentiality of *received* messages. This is a strange choice, considering it is simultaneously assumed that the adversary has complete control of all network traffic, and thus could easily decide not to deliver a message. It has long been dismissed as an unavoidable attack but we identify that the reason for this is the lack of modelling of physical time, which is required to express that ciphertexts may time out and expire. Hence we developed new security models for the out-of-order delivery setting with immediate decryption. Our model incorporates the concept of physical clocks and implements a maximally strong corruption model. In Sec. 2.7 we discussed some questions the formal treatment of Chapter 2 may have raised and elaborated on the thought process behind the design choices.

A large part of this thesis has focused on securing the content of instant messages. However, as alluded to earlier, this all feels irrelevant if the contents are subsequently uploaded to the cloud in plaintext. It is important to recognize that in practice no system is used in isolation and we have looked at secure backup solutions as a step towards bridging the gap between theory and practice in the field of instant messaging. We explored techniques to enable devices to securely outsource storage of data in Chapter 3.

We identified that in our case, where encryptor and decryptor are the same entity, we can maintain integrity protection when the key is compromised. We introduced the ETS primitive and designed a surprisingly efficient provably secure construction of ETS. Since its design choices were already extensively discussed during its description, we will not repeat that discussion here. Our ETS construction assumes a random key is provided. In Chapter 4 we demonstrated how a user can actually generate (and recover) a cryptographic key to decrypt its backup after losing access to its device. Our solutions rely on the existence of OPRF servers to convert a human-memorable password into a single cryptographic secret. In Sec. 4.8 we commented on some of the design choices made when formalizing our backup solution and we discussed some suggestions for future research in the area.

References

[1] Michel Abdalla, Mario Cornejo, Anca Nitulescu, and David Pointcheval. "Robust Password-Protected Secret Sharing". In: *ESORICS 2016, Part II*. Ed. by Ioannis G. Askoxylakis, Sotiris Ioannidis, Sokratis K. Katsikas, and Catherine A. Meadows. Vol. 9879. LNCS. Springer, Heidelberg, Sept. 2016, pp. 61–79. DOI: 10.1007/978-3-319-45741-3_4.

[2] Martin Albrecht, Lenka Mareková, Kenny Paterson, and Igors Stepanovs. "Four Attacks and a Proof for Telegram". In: *43rd IEEE Symposium on Security and Privacy*. Vol. 1. 2022, pp. 223–242.

[3] Martin R. Albrecht, Alex Davidson, Amit Deo, and Nigel P. Smart. "Round-Optimal Verifiable Oblivious Pseudorandom Functions from Ideal Lattices". In: *PKC 2021, Part II*. Ed. by Juan Garay. Vol. 12711. LNCS. Springer, Heidelberg, May 2021, pp. 261–289. DOI: 10.1007/978-3-030-75248-4_10.

[4] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. "CoCoA: Concurrent Continuous Group Key Agreement". In: *EUROCRYPT 2022*. 2022.

[5] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. "The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol". In: *EUROCRYPT 2019, Part I*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11476. LNCS. Springer, Heidelberg, May 2019, pp. 129–158. DOI: 10.1007/978-3-030-17653-2_5.

[6] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. "Modular Design of Secure Group Messaging Protocols and the Security of MLS". In: *ACM CCS 2021*. Ed. by Giovanni Vigna and Elaine Shi. ACM Press, Nov. 2021, pp. 1463–1483. DOI: 10.1145/3460120.3484820.

[7] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. "Security Analysis and Improvements for the IETF MLS Standard for Group Messaging". In: *CRYPTO 2020, Part I*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12170. LNCS. Springer, Heidelberg, Aug. 2020, pp. 248–277. DOI: 10.1007/978-3-030-56784-2_9.

[8] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. "Continuous Group Key Agreement with Active Security". In: *TCC 2020, Part II*. Ed. by Rafael Pass and Krzysztof Pietrzak. Vol. 12551. LNCS. Springer, Heidelberg, Nov. 2020, pp. 261–290. DOI: 10.1007/978-3-030-64378-2_10.

[9] Jean-Philippe Aumasson, Samuel Neves, Zooko Wilcox-O'Hearn, and Christian Winnerlein. "BLAKE2: Simpler, Smaller, Fast as MD5". In: *ACNS 13*. Ed. by Michael J. Jacobson Jr., Michael E. Locasto, Payman Mohassel, and Reihaneh Safavi-Naini. Vol. 7954. LNCS. Springer, Heidelberg, June 2013, pp. 119–135. DOI: 10.1007/978-3-642-38980-1_8.

[10] Christoph Bader, Tibor Jager, Yong Li, and Sven Schäge. "On the Impossibility of Tight Cryptographic Reductions". In: *EUROCRYPT 2016, Part II*. Ed. by Marc Fischlin and Jean-Sébastien Coron. Vol. 9666. LNCS. Springer, Heidelberg, May 2016, pp. 273–304. DOI: 10.1007/978-3-662-49896-5_10.

[11] Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. "Password-protected secret sharing". In: *ACM CCS 2011*. Ed. by Yan Chen, George Danezis, and Vitaly Shmatikov. ACM Press, Oct. 2011, pp. 433–444. DOI: 10.1145/2046707.2046758.

[12] Fatih Balli, Paul Rösler, and Serge Vaudenay. "Determining the Core Primitive for Optimally Secure Ratcheting". In: *ASIACRYPT 2020, Part III*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12493. LNCS. Springer, Heidelberg, Dec. 2020, pp. 621–650. DOI: 10.1007/978-3-030-64840-4_21.

[13] Abhishek Banerjee and Chris Peikert. "New and Improved Key-Homomorphic Pseudorandom Functions". In: *CRYPTO 2014, Part I*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8616. LNCS. Springer, Heidelberg, Aug. 2014, pp. 353–370. DOI: 10.1007/978-3-662-44371-2_20.

[14] Andrea Basso, Péter Kutas, Simon-Philipp Merz, Christophe Petit, and Antonio Sanso. "Cryptanalysis of an Oblivious PRF from Supersingular Isogenies". In: *ASIACRYPT 2021, Part I*. Ed. by Mehdi Tibouchi and Huaxiong Wang. Vol. 13090. LNCS. Springer, Heidelberg, Dec. 2021, pp. 160–184. DOI: 10.1007/978-3-030-92062-3_6.

[15] Carsten Baum, Tore Kasper Frederiksen, Julia Hesse, Anja Lehmann, and Avishay Yanai. "PESTO: Proactively Secure Distributed Single Sign-On, or How to Trust a Hacked Server". In: *IEEE EuroS&P 2020*. Ed. by Lujo Bauer and Frank Stajano. IEEE, 2020, pp. 587–606. DOI: 10.1109/EuroSP48549.2020.00044. URL: https://doi.org/10.1109/EuroSP48549.2020.00044.

[16] Mihir Bellare, Rafael Dowsley, Brent Waters, and Scott Yilek. "Standard Security Does Not Imply Security against Selective-Opening". In: *EUROCRYPT 2012*. Ed. by David Pointcheval and Thomas Johansson. Vol. 7237. LNCS. Springer, Heidelberg, Apr. 2012, pp. 645–662. DOI: 10.1007/978-3-642-29011-4_38.

[17] Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. "Message-Locked Encryption and Secure Deduplication". In: *EUROCRYPT 2013*. Ed. by Thomas Johansson and Phong Q. Nguyen. Vol. 7881. LNCS. Springer, Heidelberg, May 2013, pp. 296–312. DOI: 10.1007/978-3-642-38348-9_18.

[18] Mihir Bellare, Tadayoshi Kohno, and Chanathip Namprempre. "Authenticated Encryption in SSH: Provably Fixing The SSH Binary Packet Protocol". In: *ACM CCS 2002*. Ed. by Vijayalakshmi Atluri. ACM Press, Nov. 2002, pp. 1–11. DOI: 10.1145/586110.586112.

[19] Mihir Bellare and Sara K. Miner. "A Forward-Secure Digital Signature Scheme". In: *CRYPTO'99*. Ed. by Michael J. Wiener. Vol. 1666. LNCS. Springer, Heidelberg, Aug. 1999, pp. 431–448. DOI: 10.1007/3-540-48405-1_28.

[20] Fabrice Benhamouda and David Pointcheval. *Verifier-Based Password-Authenticated Key Exchange: New Models and Constructions*. Cryptology ePrint Archive, Report 2013/833. https://eprint.iacr.org/2013/833. 2013.

[21] Eli Biham and Rafi Chen. "Near-Collisions of SHA-0". In: *CRYPTO 2004*. Ed. by Matthew Franklin. Vol. 3152. LNCS. Springer, Heidelberg, Aug. 2004, pp. 290–305. DOI: 10.1007/978-3-540-28628-8_18.

[22] Eli Biham and Orr Dunkelman. *A Framework for Iterative Hash Functions - HAIFA*. Cryptology ePrint Archive, Report 2007/278. https://eprint.iacr.org/2007/278. 2007.

[23] Alex Biryukov, Daniel Dinu, and Dmitry Khovratovich. "Argon2: New Generation of Memory-Hard Functions for Password Hashing and Other Applications". In: *EuroS&P*. IEEE, 2016, pp. 292–302.

[24] Dan Boneh and Xavier Boyen. "Efficient Selective-ID Secure Identity Based Encryption Without Random Oracles". In: *EUROCRYPT 2004*. Ed. by Christian Cachin and Jan Camenisch. Vol. 3027. LNCS. Springer, Heidelberg, May 2004, pp. 223–238. DOI: 10.1007/978-3-540-24676-3_14.

[25] Dan Boneh, Dmitry Kogan, and Katharine Woo. "Oblivious Pseudorandom Functions from Isogenies". In: *ASIACRYPT 2020, Part II*. Ed. by Shiho Moriai and Huaxiong Wang. Vol. 12492. LNCS. Springer, Heidelberg, Dec. 2020, pp. 520–550. DOI: 10.1007/978-3-030-64834-3_18.

[26] Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. "Key Homomorphic PRFs and Their Applications". In: *CRYPTO 2013, Part I*. Ed. by Ran Canetti and Juan A. Garay. Vol. 8042. LNCS. Springer, Heidelberg, Aug. 2013, pp. 410–428. DOI: 10.1007/978-3-642-40041-4_23.

[27] Dan Boneh and Victor Shoup. *A graduate course in applied cryptography*. 2020.

[28] Daniel Bourdrez, Dr. Hugo Krawczyk, Kevin Lewi, and Christopher A. Wood. *The OPAQUE Asymmetric PAKE Protocol*. Internet-Draft draft-irtf-cfrg-opaque-08. Work in Progress. Internet Engineering Task Force, Mar. 2022. 67 pp. URL: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-opaque-08.

[29] Colin Boyd, Gareth T. Davies, Kristian Gjøsteen, and Yao Jiang. "Fast and Secure Updatable Encryption". In: *CRYPTO 2020, Part I*. Ed. by Daniele Micciancio and Thomas Ristenpart. Vol. 12170. LNCS. Springer, Heidelberg, Aug. 2020, pp. 464–493. DOI: 10.1007/978-3-030-56784-2_16.

[30] Emmanuel Bresson, Olivier Chevassut, and David Pointcheval. "New Security Results on Encrypted Key Exchange". In: *PKC 2004*. Ed. by Feng Bao, Robert Deng, and Jianying Zhou. Vol. 2947. LNCS. Springer, Heidelberg, Mar. 2004, pp. 145–158. DOI: 10.1007/978-3-540-24632-9_11.

[31] Julian Brost, Christoph Egger, Russell W. F. Lai, Fritz Schmid, Dominique Schröder, and Markus Zoppelt. "Threshold Password-Hardened Encryption Services". In: *ACM CCS 2020*. Ed. by Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna. ACM Press, Nov. 2020, pp. 409–424. DOI: 10.1145/3372297.3417266.

[32] Andrea Caforio, F Betül Durak, and Serge Vaudenay. *On-Demand Ratcheting with Security Awareness*. Cryptology ePrint Archive, Report 2019/965. https://eprint.iacr.org/2019/965. 2019.

[33] Jan Camenisch, Anja Lehmann, Anna Lysyanskaya, and Gregory Neven. "Memento: How to Reconstruct Your Secrets from a Single Password in a Hostile Environment". In: *CRYPTO 2014, Part II*. Ed. by Juan A. Garay and Rosario Gennaro. Vol. 8617. LNCS. Springer, Heidelberg, Aug. 2014, pp. 256–275. DOI: 10.1007/978-3-662-44381-1_15.

[34] Jan Camenisch, Anna Lysyanskaya, and Gregory Neven. "Practical yet universally composable two-server password-authenticated secret sharing". In: *ACM CCS 2012*. Ed. by Ting Yu, George Danezis, and Virgil D. Gligor. ACM Press, Oct. 2012, pp. 525–536. DOI: 10.1145/2382196.2382252.

[35] Ran Canetti. "Universally Composable Security: A New Paradigm for Cryptographic Protocols". In: *42nd FOCS*. IEEE Computer Society Press, Oct. 2001, pp. 136–145. DOI: 10.1109/SFCS.2001.959888.

[36] Ran Canetti, Shai Halevi, and Jonathan Katz. "A Forward-Secure Public-Key Encryption Scheme". In: *EUROCRYPT 2003*. Ed. by Eli Biham. Vol. 2656. LNCS. Springer, Heidelberg, May 2003, pp. 255–271. DOI: 10.1007/3-540-39200-9_16.

[37] Sílvia Casacuberta, Julia Hesse, and Anja Lehmann. "SoK: Oblivious Pseudorandom Functions". In: *IEEE EuroS&P 2022*. Ed. by David Evans and Carmela Troncoso. IEEE, 2022.

[38] Wouter Castryck, Tanja Lange, Chloe Martindale, Lorenz Panny, and Joost Renes. "CSIDH: An Efficient Post-Quantum Commutative Group Action". In: *ASIACRYPT 2018, Part III*. Ed. by Thomas Peyrin and Steven Galbraith. Vol. 11274. LNCS. Springer, Heidelberg, Dec. 2018, pp. 395–427. DOI: 10.1007/978-3-030-03332-3_15.

[39] Donghoon Chang, Mridul Nandi, and Moti Yung. *Indifferentiability of the Hash Algorithm BLAKE*. Cryptology ePrint Archive, Report 2011/623. https://eprint.iacr.org/2011/623. 2011.

[40] Katriel Cohn-Gordon, Cas Cremers, Benjamin Dowling, Luke Garratt, and Douglas Stebila. "A Formal Security Analysis of the Signal Messaging Protocol". In: *2017 IEEE European Symposium on Security and Privacy (EuroS&P)*. 2017, pp. 451–466.

[41] Poulami Das, Julia Hesse, and Anja Lehmann. "DPaSE: Distributed Password-Authenticated Symmetric-Key Encryption, or How to Get Many Keys from One Password". In: *ASIACCS 22*. Ed. by Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako. ACM Press, May 2022, pp. 682–696. DOI: 10.1145/3488932.3517389.

[42] Alex Davidson, Armando Faz-Hernández, Nick Sullivan, and Christopher A. Wood. *Oblivious Pseudorandom Functions (OPRFs) using Prime-Order Groups*. Internet-Draft draft-irtf-cfrg-voprf-09. Work in Progress. Internet Engineering Task Force, Feb. 2022. 63 pp. URL: https://datatracker.ietf.org/doc/html/draft-irtf-cfrg-voprf-09.

[43] Alex Davidson, Ian Goldberg, Nick Sullivan, George Tankersley, and Filippo Valsorda. "Privacy Pass: Bypassing Internet Challenges Anonymously". In: *PoPETs* 2018.3 (July 2018), pp. 164–180. DOI: 10.1515/popets-2018-0026.

[44] Gareth T. Davies and Jeroen Pijnenburg. "PERKS: Persistent and Distributed Key Acquisition for Secure Storage from Passwords". In: *SAC 2022*. LNCS. Springer, Heidelberg, Aug. 2022, To Appear.

[45] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. "Fast Message Franking: From Invisible Salamanders to Encryptment". In: *CRYPTO 2018, Part I*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10991. LNCS. Springer, Heidelberg, Aug. 2018, pp. 155–186. DOI: 10.1007/978-3-319-96884-1_6.

[46] Benjamin Dowling, Douglas Stebila, and Greg Zaverucha. "Authenticated Network Time Synchronization". In: *USENIX Security 2016*. Ed. by Thorsten Holz and Stefan Savage. USENIX Association, Aug. 2016, pp. 823–840.

[47] Rafael Dowsley, Jeroen van de Graaf, Jörn Müller-Quade, and Anderson C. A. Nascimento. "Oblivious Transfer Based on the McEliece Assumptions". In: *ICITS 08*. Ed. by Reihaneh Safavi-Naini. Vol. 5155. LNCS. Springer, Heidelberg, Aug. 2008, pp. 107–117. DOI: 10.1007/978-3-540-85093-9_11.

[48] F. Betül Durak and Serge Vaudenay. "Bidirectional Asynchronous Ratcheted Key Agreement with Linear Complexity". In: *IWSEC 19*. Ed. by Nuttapong Attrapadung and Takeshi Yagi. Vol. 11689. LNCS. Springer, Heidelberg, Aug. 2019, pp. 343–362. DOI: 10.1007/978-3-030-26834-3_20.

[49] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. "The Pythia PRF Service". In: *USENIX Security 2015*. Ed. by Jaeyeon Jung and Thorsten Holz. USENIX Association, Aug. 2015, pp. 547–562.

[50] Adam Everspaugh, Rahul Chatterjee, Samuel Scott, Ari Juels, and Thomas Ristenpart. *The Pythia PRF Service*. Cryptology ePrint Archive, Report 2015/644. https://eprint.iacr.org/2015/644. 2015.

[51] Michael J. Freedman, Yuval Ishai, Benny Pinkas, and Omer Reingold. "Keyword Search and Oblivious Pseudorandom Functions". In: *TCC 2005*. Ed. by Joe Kilian. Vol. 3378. LNCS. Springer, Heidelberg, Feb. 2005, pp. 303–324. DOI: 10.1007/978-3-540-30576-7_17.

[52] Craig Gentry and Alice Silverberg. "Hierarchical ID-Based Cryptography". In: *ASIACRYPT 2002*. Ed. by Yuliang Zheng. Vol. 2501. LNCS. Springer, Heidelberg, Dec. 2002, pp. 548–566. DOI: 10.1007/3-540-36178-2_34.

[53] Federico Giacon, Felix Heuer, and Bertram Poettering. "KEM Combiners". In: *PKC 2018, Part I*. Ed. by Michel Abdalla and Ricardo Dahab. Vol. 10769. LNCS. Springer, Heidelberg, Mar. 2018, pp. 190–218. DOI: 10.1007/978-3-319-76578-5_7.

[54] Shay Gueron. “Memory Encryption for General-Purpose Processors”. In: *IEEE Secur. Priv.* 14.6 (2016), pp. 54–62.

[55] Sharon Huang et al. *DIT: Deidentified authenticated telemetry at scale*. Blog Post. Meta, Apr. 2021. URL: https://engineering.fb.com/2021/04/16/production-engineering/dit/.

[56] Joseph Jaeger and Igors Stepanovs. “Optimal Channel Security Against Fine-Grained State Compromise: The Safety of Messaging”. In: *CRYPTO 2018, Part I*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10991. LNCS. Springer, Heidelberg, Aug. 2018, pp. 33–62. DOI: 10.1007/978-3-319-96884-1_2.

[57] David Jao and Luca De Feo. “Towards Quantum-Resistant Cryptosystems from Supersingular Elliptic Curve Isogenies”. In: *Post-Quantum Cryptography - 4th International Workshop, PQCrypto 2011*. Ed. by Bo-Yin Yang. Springer, Heidelberg, Nov. 2011, pp. 19–34. DOI: 10.1007/978-3-642-25405-5_2.

[58] Stanislaw Jarecki, Aggelos Kiayias, and Hugo Krawczyk. “Round-Optimal Password-Protected Secret Sharing and T-PAKE in the Password-Only Model”. In: *ASIACRYPT 2014, Part II*. Ed. by Palash Sarkar and Tetsu Iwata. Vol. 8874. LNCS. Springer, Heidelberg, Dec. 2014, pp. 233–253. DOI: 10.1007/978-3-662-45608-8_13.

[59] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. “Highly-Efficient and Composable Password-Protected Secret Sharing (Or: How to Protect Your Bitcoin Wallet Online)”. In: *IEEE European Symposium on Security and Privacy, EuroS&P 2016*. Ed. by Michael Backes. IEEE, 2016, pp. 276–291. DOI: 10.1109/EuroSP.2016.30. URL: https://doi.org/10.1109/EuroSP.2016.30.

[60] Stanislaw Jarecki, Aggelos Kiayias, Hugo Krawczyk, and Jiayu Xu. “TOPPSS: Cost-Minimal Password-Protected Secret Sharing Based on Threshold OPRF”. In: *ACNS 17*. Ed. by Dieter Gollmann, Atsuko Miyaji, and Hiroaki Kikuchi. Vol. 10355. LNCS. Springer, Heidelberg, July 2017, pp. 39–58. DOI: 10.1007/978-3-319-61204-1_3.

[61] Stanislaw Jarecki, Hugo Krawczyk, and Jason Resch. *Threshold Partially-Oblivious PRFs with Applications to Key Management*. Cryptology ePrint Archive, Report 2018/733. https://eprint.iacr.org/2018/733. 2018.

[62] Stanislaw Jarecki, Hugo Krawczyk, and Jason K. Resch. “Updatable Oblivious Key Management for Storage Systems”. In: *ACM CCS 2019*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM Press, Nov. 2019, pp. 379–393. DOI: 10.1145/3319535.3363196.

[63] Stanislaw Jarecki, Hugo Krawczyk, and Jiayu Xu. “OPAQUE: An Asymmetric PAKE Protocol Secure Against Pre-computation Attacks”. In: *EUROCRYPT 2018, Part III*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10822. LNCS. Springer, Heidelberg, Apr. 2018, pp. 456–486. DOI: 10.1007/978-3-319-78372-7_15.

[64] Stanislaw Jarecki and Xiaomin Liu. “Efficient Oblivious Pseudorandom Function with Applications to Adaptive OT and Secure Computation of Set Intersection”. In: *TCC 2009*. Ed. by Omer Reingold. Vol. 5444. LNCS. Springer, Heidelberg, Mar. 2009, pp. 577–594. DOI: 10.1007/978-3-642-00457-5_34.

[65] Daniel Jost, Ueli Maurer, and Marta Mularczyk. "A Unified and Composable Take on Ratcheting". In: *TCC 2019, Part II*. Ed. by Dennis Hofheinz and Alon Rosen. Vol. 11892. LNCS. Springer, Heidelberg, Dec. 2019, pp. 180–210. DOI: 10.1007/978-3-030-36033-7_7.

[66] Daniel Jost, Ueli Maurer, and Marta Mularczyk. "Efficient Ratcheting: Almost-Optimal Guarantees for Secure Messaging". In: *EUROCRYPT 2019, Part I*. Ed. by Yuval Ishai and Vincent Rijmen. Vol. 11476. LNCS. Springer, Heidelberg, May 2019, pp. 159–188. DOI: 10.1007/978-3-030-17653-2_6.

[67] Burt Kaliski. *PKCS #5: Password-Based Cryptography Specification Version 2.0*. RFC 2898. Sept. 2000. DOI: 10.17487/RFC2898. URL: https://rfc-editor.org/rfc/rfc2898.txt.

[68] Sriram Keelveedhi, Mihir Bellare, and Thomas Ristenpart. "DupLESS: Server-Aided Encryption for Deduplicated Storage". In: *USENIX Security 2013*. Ed. by Samuel T. King. USENIX Association, Aug. 2013, pp. 179–194.

[69] Vladimir Kolesnikov, Ranjit Kumaresan, Mike Rosulek, and Ni Trieu. "Efficient Batched Oblivious PRF with Applications to Private Set Intersection". In: *ACM CCS 2016*. Ed. by Edgar R. Weippl, Stefan Katzenbeisser, Christopher Kruegel, Andrew C. Myers, and Shai Halevi. ACM Press, Oct. 2016, pp. 818–829. DOI: 10.1145/2976749.2978381.

[70] Russell W. F. Lai, Christoph Egger, Manuel Reinert, Sherman S. M. Chow, Matteo Maffei, and Dominique Schröder. "Simple Password-Hardened Encryption Services". In: *USENIX Security 2018*. Ed. by William Enck and Adrienne Porter Felt. USENIX Association, Aug. 2018, pp. 1405–1421.

[71] Anja Lehmann. "ScrambleDB: Oblivious (Chameleon) Pseudonymization-as-a-Service". In: *PoPETs* 2019.3 (July 2019), pp. 289–309. DOI: 10.2478/popets-2019-0048.

[72] Anja Lehmann and Björn Tackmann. "Updatable Encryption with Post-Compromise Security". In: *EUROCRYPT 2018, Part III*. Ed. by Jesper Buus Nielsen and Vincent Rijmen. Vol. 10822. LNCS. Springer, Heidelberg, Apr. 2018, pp. 685–716. DOI: 10.1007/978-3-319-78372-7_22.

[73] Allison B. Lewko and Brent Waters. "Unbounded HIBE and Attribute-Based Encryption". In: *EUROCRYPT 2011*. Ed. by Kenneth G. Paterson. Vol. 6632. LNCS. Springer, Heidelberg, May 2011, pp. 547–567. DOI: 10.1007/978-3-642-20465-4_30.

[74] C. Li and B. Palanisamy. "Timed-Release of Self-Emerging Data Using Distributed Hash Tables". In: *2017 IEEE 37th International Conference on Distributed Computing Systems (ICDCS)*. 2017, pp. 2344–2351.

[75] Lucy Li, Bijeeta Pal, Junade Ali, Nick Sullivan, Rahul Chatterjee, and Thomas Ristenpart. "Protocols for Checking Compromised Credentials". In: *ACM CCS 2019*. Ed. by Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz. ACM Press, Nov. 2019, pp. 1387–1403. DOI: 10.1145/3319535.3354229.

[76] Jia Liu, Tibor Jager, Saqib A Kakvi, and Bogdan Warinschi. "How to build time-lock encryption". In: *Designs, Codes and Cryptography* 86.11 (2018), pp. 2549–2586.

[77] M. Marlinspike and T. Perrin. *The Double Ratchet Algorithm*. Nov. 2016. URL: https://signal.org/docs/specifications/doubleratchet/doubleratchet.pdf.

[78] Giorgia Azzurra Marson and Bertram Poettering. "Security Notions for Bidirectional Channels". In: *IACR Trans. Symm. Cryptol.* 2017.1 (2017), pp. 405–426. ISSN: 2519-173X. DOI: 10.13154/tosc.v2017.i1.405-426.

[79] NCC-Group. *End-to-End Encrypted Backups Security Assessment: WhatsApp (version 1.2)*. https://research.nccgroup.com/wp-content/uploads/2021/10/NCC_Group_WhatsApp_E001000M_Report_2021-10-27_v1.2.pdf. Oct. 2021.

[80] NIST. *FIPS 180-4: Secure Hash Standard (SHS)*. Tech. rep. NIST, 2015. URL: http://dx.doi.org/10.6028/NIST.FIPS.180-4.

[81] Chris Peikert, Vinod Vaikuntanathan, and Brent Waters. "A Framework for Efficient and Composable Oblivious Transfer". In: *CRYPTO 2008*. Ed. by David Wagner. Vol. 5157. LNCS. Springer, Heidelberg, Aug. 2008, pp. 554–571. DOI: 10.1007/978-3-540-85174-5_31.

[82] Jeroen Pijnenburg and Bertram Poettering. "Efficiency Improvements for Encrypt-to-Self". In: *CYSARM@CCS*. ACM, Nov. 2020, pp. 13–23. DOI: 10.1145/3411505.3418438.

[83] Jeroen Pijnenburg and Bertram Poettering. "Encrypt-to-Self: Securely Outsourcing Storage". In: *ESORICS 2020, Part I*. Ed. by Liqun Chen, Ninghui Li, Kaitai Liang, and Steve A. Schneider. Vol. 12308. LNCS. Springer, Heidelberg, Sept. 2020, pp. 635–654. DOI: 10.1007/978-3-030-58951-6_31.

[84] Jeroen Pijnenburg and Bertram Poettering. "Key Assignment Schemes with Authenticated Encryption, revisited". In: *IACR Trans. Symm. Cryptol.* 2020.2 (2020), pp. 40–67. ISSN: 2519-173X. DOI: 10.13154/tosc.v2020.i2.40-67.

[85] Jeroen Pijnenburg and Bertram Poettering. "On Secure Ratcheting with Immediate Decryption". In: *ASIACRYPT 2022, Part III*. Ed. by Shweta Agrawal and Dongdai Lin. Vol. 13793. Lecture Notes in Computer Science. Springer, Dec. 2022, pp. 89–118. DOI: 10.1007/978-3-031-22969-5_4.

[86] Bertram Poettering and Paul Rösler. "Towards Bidirectional Ratcheted Key Exchange". In: *CRYPTO 2018, Part I*. Ed. by Hovav Shacham and Alexandra Boldyreva. Vol. 10991. LNCS. Springer, Heidelberg, Aug. 2018, pp. 3–32. DOI: 10.1007/978-3-319-96884-1_1.

[87] Phillip Rogaway. "Authenticated-Encryption With Associated-Data". In: *ACM CCS 2002*. Ed. by Vijayalakshmi Atluri. ACM Press, Nov. 2002, pp. 98–107. DOI: 10.1145/586110.586125.

[88] Phillip Rogaway. "Nonce-Based Symmetric Encryption". In: *FSE 2004*. Ed. by Bimal K. Roy and Willi Meier. Vol. 3017. LNCS. Springer, Heidelberg, Feb. 2004, pp. 348–359. DOI: 10.1007/978-3-540-25937-4_22.

[89] Markku-Juhani O. Saarinen and Jean-Philippe Aumasson. *The BLAKE2 Cryptographic Hash and Message Authentication Code (MAC)*. RFC 7693. 2015. DOI: 10.17487/RFC7693. URL: https://rfc-editor.org/rfc/rfc7693.txt.

[90] Jörg Schwenk. "Modelling Time for Authenticated Key Exchange Protocols". In: *ESORICS 2014, Part II*. Ed. by Miroslaw Kutylowski and Jaideep Vaidya. Vol. 8713. LNCS. Springer, Heidelberg, Sept. 2014, pp. 277–294. DOI: 10.1007/978-3-319-11212-1_16.

[91] István András Seres, Máté Horváth, and Péter Burcsi. *The Legendre Pseudorandom Function as a Multivariate Quadratic Cryptosystem: Security and Applications*. Cryptology ePrint Archive, Report 2021/182. https://eprint.iacr.org/2021/182. 2021.

[92] Adi Shamir. "How to Share a Secret". In: *Communications of the Association for Computing Machinery* 22.11 (Nov. 1979), pp. 612–613.

[93] Mark W. Storer, Kevin M. Greenan, Darrell D. E. Long, and Ethan L. Miller. "Secure data deduplication". In: *StorageSS 2008*. Ed. by Yongdae Kim and William Yurcik. ACM, 2008, pp. 1–10. DOI: 10.1145/1456469.1456471. URL: https://doi.org/10.1145/1456469.1456471.

[94] Nick Sullivan, Dr. Hugo Krawczyk, Owen Friel, and Richard Barnes. *OPAQUE with TLS 1.3*. Internet-Draft draft-sullivan-tls-opaque-01. Work in Progress. Internet Engineering Task Force, Feb. 2021. 13 pp. URL: https://datatracker.ietf.org/doc/html/draft-sullivan-tls-opaque-01.

[95] Kurt Thomas et al. "Protecting accounts from credential stuffing with password breach alerting". In: *USENIX Security 2019*. Ed. by Nadia Heninger and Patrick Traynor. USENIX Association, Aug. 2019, pp. 1556–1571.

[96] Nirvan Tyagi, Sofía Celi, Thomas Ristenpart, Nick Sullivan, Stefano Tessaro, and Christopher A. Wood. "A Fast and Simple Partially Oblivious PRF, with Applications". In: *EUROCRYPT 2022, Part II*. Ed. by Orr Dunkelman and Stefan Dziembowski. Vol. 13276. LNCS. Springer, Heidelberg, May 2022, pp. 674–705. DOI: 10.1007/978-3-031-07085-3_23.

[97] WhatsApp. *Security of End-To-End Encrypted Backups*. https://www.whatsapp.com/security/WhatsApp_Security_Encrypted_Backups_Whitepaper.pdf. Sept. 2021.

[98] Hailun Yan and Serge Vaudenay. "Symmetric Asynchronous Ratcheted Communication with Associated Data". In: *IWSEC 20*. Ed. by Kazumaro Aoki and Akira Kanaoka. Vol. 12231. LNCS. Springer, Heidelberg, Sept. 2020, pp. 184–204. DOI: 10.1007/978-3-030-58208-1_11.