A preliminary version of this paper appears in CRYPTO 2018. This is the full, fixed version.

Optimal Channel Security Against Fine-Grained State Compromise: The Safety of Messaging

Joseph Jaeger[1] Igors Stepanovs[2]

August 2018

Abstract

We aim to understand the best possible security of a (bidirectional) cryptographic channel against an adversary that may arbitrarily and repeatedly learn the secret state of either communicating party. We give a formal security definition and a proven-secure construction. This construction provides better security against state compromise than the Signal Double Ratchet Algorithm or any other known channel construction. To facilitate this we define and construct new forms of public-key encryption and digital signatures that update their keys over time.

[1] Department of Computer Science & Engineering, University of California San Diego, 9500 Gilman Drive, La Jolla, California 92093, USA. Email: jsjaeger@eng.ucsd.edu. URL: https://cseweb.ucsd.edu/~jsjaeger/. Supported in part by NSF grants CNS-1717640 and CNS-1526801.

[2] Department of Computer Science & Engineering, University of California San Diego, 9500 Gilman Drive, La Jolla, California 92093, USA. Email: istepano@eng.ucsd.edu. URL: https://cseweb.ucsd.edu/~istepano/. Supported in part by NSF grants CNS-1717640 and CNS-1526801.

Contents

1 Introduction

End-to-end encrypted communication is becoming a usable reality for the masses in the form of secure messaging apps. However, chat sessions can be extremely long-lived and their secrets are stored on end user devices, so they are particularly vulnerable to having their cryptographic secrets exfiltrated to an attacker by malware or physical access to the device. The Signal protocol [34] by Open Whisper Systems tries to mitigate this threat by continually updating the key used for encryption. Beyond its use in the Signal messaging app, this protocol has been adopted by a number of other secure messaging apps. This includes being used by default in WhatsApp and as part of secure messaging modes of Facebook Messenger, Google Allo, and Skype.

WhatsApp alone has 1 billion daily active users [44]. It is commonly agreed in the cryptography and security community that the Signal protocol is secure. However, the protocol was designed without an explicitly defined security notion. This raises the questions: what security does it achieve and could we do better?

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

Security against compromise. When a party's secret state is exposed we would like both that the security of past messages and (as soon as possible) the security of future messages not be damaged. These notions have been considered in a variety of contexts with differing terminology. The systemization of knowledge paper on secure messaging [43] by Unger, Dechand, Bonneau, Fahl, Perl, Goldberg, and Smith evaluates and systematizes a number of secure messaging systems. In it they describe a variety of terms for these types of security including "forward secrecy," "backwards secrecy," "self-healing," and "future secrecy" and note that they are "controversial and vague." Cohn-Gordon, Cremers, and Garratt [17] study the future direction under the term of post-compromise security and similarly discuss the terms "future secrecy," "healing," and "bootstrapping" and note that they are "intuitive" but "not well-defined." Our security notion intuitively captures any of these informal terms, but we avoid using any of them directly by aiming generically for the best possible security against compromise.

Channels. The standard model for studying secure two party communication is that of the (cryptographic) channel. The first attempts to consider the secure channel as a cryptographic object were made by Shoup [40] and Canetti [13]. It was then formalized by Canetti and Krawczyk [15] as a modular way to combine a key exchange protocol with authenticated encryption, which covers both privacy and integrity. Krawczyk [29] and Namprempre [33] study what are the necessary and sufficient security notions to build a secure channel from these primitives.

Modern definitions of channels often draw from the game-based notion of security for stateful authenticated-encryption as defined by Bellare, Kohno, and Namprempre [4]. We follow this convention which assumes initial generation of keys is trusted. In addition to requiring that a channel provides integrity and privacy of the encrypted data, we will require integrity for associated data as introduced by Rogaway [37].

Recently Marson and Poettering [31] closed a gap in the modeling of two-party communication by capturing the bidirectional nature of practical channels in their definitions. We work with their notion of bidirectional channels because it closely models the behavior desired in practice and the bidirectional nature of communication allows us to achieve a fine-grained security against compromise.

Definitional contributions. This paper aims to specify and achieve the best possible security of a bidirectional channel against state compromise. We provide a formal, game-based definition of security and a construction that provably achieves it. We analyze our construction in a concrete security framework [2] and give precise bounds on the advantage of an attacker.

To derive the best possible notion of security against state compromise we first specify a basic input-output interface via a game that describes how the adversary interacts with the channel. This corresponds roughly to combining the integrity and confidentiality games of [31] and adding an oracle that returns the secret state of a specified user to the adversary. Then we specify several attacks that break the security of *any* channel. We define our final security notion by minimally extending the initial interface game to disallow these unavoidable attacks while allowing all other behaviors. Our security definition is consequently the best possible with respect to the specified interface because our attacks rule out the possibility of any stronger notion.

One security notion is an all-in-one notion in the style of [38] that simultaneously requires integrity and privacy of the channel. It asks for the maximal possible security in the face of the exposure of either party's state. A surprising requirement of our definition is that given the state of a user the adversary should not be able to decrypt ciphertexts sent by that user or send forged ciphertexts to that user.

Protocols that update their keys. The OTR (Off-the-Record) messaging protocol [12] is an important predecessor to Signal. It has parties repeatedly exchange Diffie-Hellman elements to derive new keys. The Double Ratchet Algorithm of Signal uses a similar Diffie-Hellman update mechanism and extends it by using a symmetric key-derivation function to update keys when there is no Diffie-Hellman update available. Both methods of updating keys are often referred to as ratcheting (a term introduced by Langley [30]). While the Double Ratchet Algorithm was explicitly designed to achieve strong notions of security against state compromise with respect to privacy, the designers explicitly consider security against a passive eavesdropper [23]; authenticity in the face of compromise is out of scope.

The first academic security analysis of Signal was due to Cohn-Gordan, Cremers, Dowling, Garratt, and Stebila [16]. They only considered the security of the key exchange underlying the Double Ratchet Algorithm and used a security definition explicitly tailored to understanding its security instead of being widely applicable to any scheme.

Work by Bellare, Camper Singh, Jaeger, Nyayapati, and Stepanovs [9] sought to formally understand ratcheting as an independent primitive, introducing the notions of (one-directional) ratcheted key exchange and ratcheted encryption. In their model a compromise of the receiving party's secrets permanently and irrevocably disrupts all security, past and future. Further they strictly separate the exchange of key update information from the exchange of messages. Such a model cannot capture a protocol like the Double Ratchet Algorithm for which the two are inextricably combined. On the positive side, they did explicitly model authenticity in the face of compromise.

In [28], Günther and Mazaheri study a key update mechanism introduced in TLS 1.3. Their security definition treats update messages as being out-of-band and thus implicitly authenticated. Their definition is clearly tailored to understand TLS 1.3 specifically.

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

primitives be used throughout.

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

Our construction. Our construction of a secure channel is given in Section 6.1. It shows how to generically build the channel from a collision-resistant hash function, a public-key encryption scheme, and a digital signature scheme. The latter two require new versions of the primitives that we describe momentarily.

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

New notions of public-key encryption and digital signatures. Our construction uses new forms of public-key encryption and digital signatures that update their keys over time, which we define in Section 3. They both include extra algorithms that allow the keys to be updated with respect to an arbitrary string. We refer to them as key-updatable public-key encryption and key-updatable digital signature schemes. In our secure channel construction a user updates their signing key and every decryption key they currently have with their transcript every time they receive a ciphertext.

For public-key encryption we consider encryption with labels and require an IND-CCA style security be maintained even if the adversary is given the decryption key as long as the sequence of strings used to update it is not a prefix of the sequence of strings used to update the encryption key before any challenge queries. We show that such a scheme is can be obtained directly from hierarchical identity-based encryption [25].

For digital signatures, security requires that an adversary is unable to forge a signature even given the signing key as long as the sequence of strings used to update it is not a prefix of the sequence of strings used to update the verification key. We additionally require that the scheme has unique signatures (i.e. for any sequence of updates and any message an adversary can only find one signature that will verify). We show how to construct this from a digital signature scheme that is forward secure [5] and has unique signatures.

Related work. Several works [24, 11] extended the definitions of channels to address the stream-based interface provided by channels like TLS, SSH, and QUIC. Our primary motivation is to build a channel for messaging where an atomic interface for messages is more appropriate.

Numerous areas of research within cryptography are motivated by the threat of key compromise. These include key-insulated cryptography [20, 21, 22], secret sharing [39, 32, 42], threshold cryptography [18], proactive cryptography [35], and forward security [27, 19]. Forward security, in

particular, was introduced in the context of key-exchange [27, 19] but has since been considered for a variety of primitives including symmetric [10] and asymmetric encryption [14] and digital signature schemes [5]. Green and Miers [26] propose using puncturable encryption for forward secure asynchronous messaging.

In concurrent and independent work, Poettering and Rösler [36] extend the definitions of ratcheted key exchange from [9] to be bidirectional. Their security definition is conceptually similar to our definition for bidirectional channels because both works aim to achieve strong notions of security against an adversary that can arbitrarily and repeatedly learn the secret state of either communicating party. In constructing a secure ratcheted key exchange scheme they make use of a key-updatable key encapsulation mechanism (KEM), a new primitive they introduce in their work. The key-updatable nature of this is conceptually similar to that of the key-updatable public-key encryption and digital signature schemes we introduce in our work. To construct such a KEM they make use of hierarchical identity-based encryption in a manner similar to how we construct key-updatable public-key encryption. The goal of their work differs from ours; they only consider security for the exchange of symmetric keys while we do so for the exchange of messages.

Errors in the proceedings version. The proceedings version of this paper contained a security flaw. It used a weaker variant of key-updatable public-key encryption (called "key-updating" public-key encryption). Because of this, only the oldest decryption key was updated when a ciphertext was received, instead of all stored decryption keys being updated (the latter is done by the fixed scheme presented in this version). This can be exploited to break the claimed security of the prior scheme in a straightforward manner.

2 Preliminaries

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

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

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

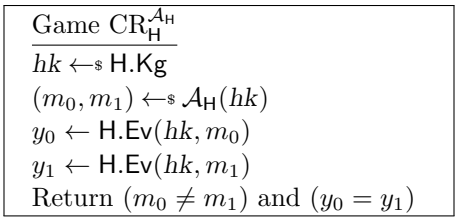

Figure 1: Game defining collision-resistance of function family H.

with output $(x_1, \dots)$.

We use a special symbol $\perp \notin \{0,1\}^*$ to denote an empty table position, and we also return it as an error code indicating an invalid input. An algorithm may not accept $\perp$ as input. If $x_i = \perp$ for some i when executing $(y_1, \dots) \leftarrow A(x_1 \dots)$ we assume that $y_j = \perp$ for all j. We assume that adversaries never pass $\perp$ as input to their oracles.

We use the code based game playing framework of [8]. (See Fig. 1 for an example of a game.) We let Pr[G] denote the probability that game G returns true. In code, tables are initially empty. We adopt the convention that the running time of an adversary means the worst case execution time of the adversary in the game that executes it, so that time for game setup steps and time to compute answers to oracle queries is included.

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

Collision-resistant functions. Consider game CR of Fig. 1 associated to a function family H and an adversary $\mathcal{A}_{\mathsf{H}}$. The game samples a random key hk for function family H. In order to win the game, adversary $\mathcal{A}_{\mathsf{H}}$ has to find two distinct messages m_0, m_1 such that $\mathsf{H.Ev}(hk, m_0) = \mathsf{H.Ev}(hk, m_1)$. The advantage of $\mathcal{A}_{\mathsf{H}}$ in breaking the CR security of H is defined as $\mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathcal{A}_{\mathsf{H}}) = \Pr[\mathrm{CR}_{\mathsf{H}}^{\mathcal{A}_{\mathsf{H}}}]$.

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

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

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

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

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

Figure 2: Games defining correctness of key-updatable digital signature scheme DS and correctness of key-updatable public-key encryption scheme PKE.

3 New asymmetric primitives

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

We specify multi-user security definitions for both primitives, because it allows tighter reductions when we construct a channel from these primitives. Single-user variants of these definitions are obtained by only allowing the adversary to interact with one user and can be shown to imply the multi-user versions by a standard hybrid argument. Starting with [1] constructions have been given for a variety of primitives that allow multi-user security to be proven without the factor q security loss introduced by a hybrid argument. If analogous constructions can be found for our primitives then our results will give tight bounds on the security of our channel.

3.1 Key-updatable digital signature schemes

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

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

Let $\vec{\Delta} = (\Delta_1, \Delta_2, \ldots, \Delta_n)$. For compactness, we sometimes write $(vk, t) \leftarrow \mathsf{DS.Vrfy}(vk, \sigma, m, \vec{\Delta})$ to denote updating the verification key via $vk \leftarrow \mathsf{DS.UpdVk}(vk, \Delta_i)$ for $i = 1, \ldots, n$ and then

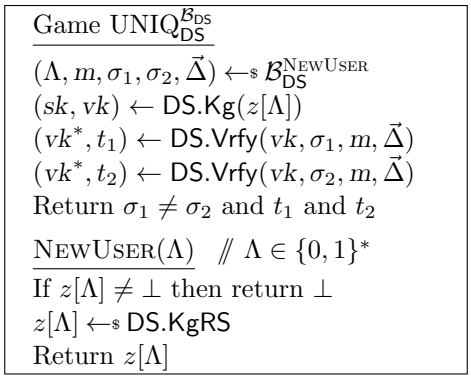

Figure 3: Game defining signature uniqueness of key-updatable digital signature scheme DS.

evaluating $t \leftarrow \mathsf{DS.Vrfy}(vk, \sigma, m)$.

The key-update correctness condition requires that signatures must verify correctly as long as the signing and the verification keys are both updated with the same sequence of key update information $\vec{\Delta} = (\Delta_1, \Delta_2, \ldots)$. To formalize this, consider game DSCORR of Fig. 2, associated to a key-updatable digital signature scheme DS and an adversary $\mathcal{C}$. The advantage of an adversary $\mathcal{C}$ against the correctness of DS is given by $\mathsf{Adv}^{\mathsf{dscorr}}_{\mathsf{DS}}(\mathcal{C}) = \Pr[\text{DSCORR}^{\mathcal{C}}_{\mathsf{DS}}]$. We require that $\mathsf{Adv}^{\mathsf{dscorr}}_{\mathsf{DS}}(\mathcal{C}) = 0$ for all (even unbounded) adversaries $\mathcal{C}$. See Section 4 for discussion on game-based definitions of correctness.

Signature uniqueness. We will be interested in schemes for which there is only a single signature that will be accepted for any message m and any sequence of updates $\vec{\Delta}$. Consider game UNIQ of Fig. 3, associated to a key-updatable digital signature scheme DS and an adversary $\mathcal{B}_{\mathsf{DS}}$. The adversary $\mathcal{B}_{\mathsf{DS}}$ can call the oracle NewUser arbitrarily many times with a user identifier Λ and be given the randomness used to generate the keys of Λ. The adversary ultimately outputs a user id Λ, a message m, signatures σ_1, σ_2, and a key update vector $\vec{\Delta}$. It wins if the signatures are distinct and both verify for m when the verification key of Λ is updated with $\vec{\Delta}$. The advantage of $\mathcal{B}_{\mathsf{DS}}$ in breaking the UNIQ security of DS is defined as $\mathsf{Adv}^{\mathsf{uniq}}_{\mathsf{DS}}(\mathcal{B}_{\mathsf{DS}}) = \Pr[\text{UNIQ}^{\mathcal{B}_{\mathsf{DS}}}_{\mathsf{DS}}]$.

Signature unforgeability under exposures. Our main security notion for signatures asks that the adversary not be able to create signatures for any key update vector $\vec{\Delta}$ unless it acquired a signing key for some key update vector $\vec{\Delta}'$ that is a prefix of $\vec{\Delta}$. The adversary can query an oracle to receive a signature for one message with respect to each user's secret key, but it cannot claim this signature as a forgery. Consider game UFEXP of Fig. 4, associated to a key-updatable digital signature scheme DS and an adversary $\mathcal{A}_{\mathsf{DS}}$.

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

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

Game $\mathrm{UFEXP}^{\mathcal{A}_{\mathsf{DS}}}_{\mathsf{DS}}$

$\mathsf{out} \leftarrow\!\!\$\ \mathcal{A}^{\textsc{NewUser},\textsc{Upd},\textsc{Sign},\textsc{Exp}}_{\mathsf{DS}}$
$(\Lambda, \sigma, m, \vec{\Delta}) \leftarrow \mathsf{out}$
$\mathsf{forgery} \leftarrow (\sigma, m, \vec{\Delta})$
$\mathsf{trivial} \leftarrow (\sigma^*[\Lambda], m^*[\Lambda], \vec{\Delta}^*[\Lambda])$
$t_1 \leftarrow (\mathsf{forgery} = \mathsf{trivial})$
$t_2 \leftarrow (\vec{\Delta}'[\Lambda] \sqsubseteq \vec{\Delta})$
$\mathsf{cheated} \leftarrow (t_1 \text{ or } t_2)$
$vk \leftarrow vk[\Lambda]$
$(vk, \mathsf{win}) \leftarrow \mathsf{DS.Vrfy}(vk, \sigma, m, \vec{\Delta})$
Return win and not $\mathsf{cheated}$

$\underline{\textsc{NewUser}(\Lambda)}$ // $\Lambda \in \{0,1\}^*$
If $sk[\Lambda] \neq \bot$ then return $\bot$
$\vec{\Delta}_s[\Lambda] \leftarrow ()$; $\vec{\Delta}^*[\Lambda] \leftarrow \bot$; $\vec{\Delta}'[\Lambda] \leftarrow \bot$
$(sk[\Lambda], vk[\Lambda]) \leftarrow\!\!\$\ \mathsf{DS.Kg}$
Return $vk[\Lambda]$

$\underline{\textsc{Upd}(\Lambda, \Delta)}$ // $\Lambda, \Delta \in \{0,1\}^*$
If $sk[\Lambda] = \bot$ then return $\bot$
$\vec{\Delta}_s[\Lambda] \leftarrow \vec{\Delta}_s[\Lambda] \,\|\, \Delta$
$sk[\Lambda] \leftarrow\!\!\$\ \mathsf{DS.UpdSk}(sk[\Lambda], \Delta)$
Return $\bot$

$\underline{\textsc{Sign}(\Lambda, m)}$ // $\Lambda, m \in \{0,1\}^*$
If $sk[\Lambda] = \bot$ then return $\bot$
If $\vec{\Delta}^*[\Lambda] \neq \bot$ then return $\bot$
$\sigma \leftarrow\!\!\$\ \mathsf{DS.Sign}(sk[\Lambda], m)$
$(\sigma^*[\Lambda], m^*[\Lambda], \vec{\Delta}^*[\Lambda]) \leftarrow (\sigma, m, \vec{\Delta}_s[\Lambda])$
Return σ

$\underline{\textsc{Exp}(\Lambda)}$ // $\Lambda \in \{0,1\}^*$
If $sk[\Lambda] = \bot$ then return $\bot$
If $\vec{\Delta}'[\Lambda] = \bot$ then $\vec{\Delta}'[\Lambda] \leftarrow \vec{\Delta}_s[\Lambda]$
Return $sk[\Lambda]$

Game $\mathrm{INDEXP}^{\mathcal{A}_{\mathsf{PKE}}}_{\mathsf{PKE}}$

$b \leftarrow\!\!\$\ \{0,1\}$
$b' \leftarrow\!\!\$\ \mathcal{A}^{\textsc{NewUser},\textsc{UpdEk},\textsc{UpdDk},\textsc{Enc},\textsc{Dec},\textsc{Exp}}_{\mathsf{PKE}}$
Return $b = b'$

$\underline{\textsc{NewUser}(\Lambda)}$ // $\Lambda \in \{0,1\}^*$
If $dk[\Lambda] \neq \bot$ then return $\bot$
$\vec{\Delta}_e[\Lambda] \leftarrow ()$; $\vec{\Delta}_d[\Lambda] \leftarrow ()$
$\vec{\Delta}'[\Lambda] \leftarrow \bot$; $S[\Lambda] \leftarrow \emptyset$
$(ek[\Lambda], dk[\Lambda]) \leftarrow\!\!\$\ \mathsf{PKE.Kg}$
Return $ek[\Lambda]$

$\underline{\textsc{UpdEk}(\Lambda, \Delta)}$ // $\Lambda, \Delta \in \{0,1\}^*$
If $dk[\Lambda] = \bot$ then return $\bot$
$ek[\Lambda] \leftarrow \mathsf{PKE.UpdEk}(ek[\Lambda], \Delta)$
$\vec{\Delta}_e[\Lambda] \leftarrow \vec{\Delta}_e[\Lambda] \,\|\, \Delta$

$\underline{\textsc{UpdDk}(\Lambda, \Delta)}$ // $\Lambda, \Delta \in \{0,1\}^*$
If $dk[\Lambda] = \bot$ then return $\bot$
$dk[\Lambda] \leftarrow\!\!\$\ \mathsf{PKE.UpdDk}(dk[\Lambda], \Delta)$
$\vec{\Delta}_d[\Lambda] \leftarrow \vec{\Delta}_d[\Lambda] \,\|\, \Delta$

$\underline{\textsc{Enc}(\Lambda, m_0, m_1, \ell)}$ // $\Lambda, m_0, m_1, \ell \in \{0,1\}^*$
If $dk[\Lambda] = \bot$ then return $\bot$
If $|m_0| \neq |m_1|$ then return $\bot$
If $\vec{\Delta}'[\Lambda] \sqsubseteq \vec{\Delta}_e[\Lambda]$ then return $\bot$
$c \leftarrow\!\!\$\ \mathsf{PKE.Enc}(ek[\Lambda], \ell, m_b)$
$S[\Lambda] \leftarrow S[\Lambda] \cup \{(\vec{\Delta}_e[\Lambda], c, \ell)\}$
Return c

$\underline{\textsc{Dec}(\Lambda, c, \ell)}$ // $\Lambda, c, \ell \in \{0,1\}^*$
If $dk[\Lambda] = \bot$ then return $\bot$
If $(\vec{\Delta}_d[\Lambda], c, \ell) \in S[\Lambda]$ then return $\bot$
$m \leftarrow \mathsf{PKE.Dec}(dk[\Lambda], \ell, c)$
Return m

$\underline{\textsc{Exp}(\Lambda)}$ // $\Lambda \in \{0,1\}^*$
If $dk[\Lambda] = \bot$ then return $\bot$
If $\exists (\vec{\Delta}, c, \ell) \in S[\Lambda]$ s.t. $\vec{\Delta}_d[\Lambda] \sqsubseteq \vec{\Delta}$ then
  Return $\bot$
If $\vec{\Delta}'[\Lambda] = \bot$ then $\vec{\Delta}'[\Lambda] \leftarrow \vec{\Delta}_d[\Lambda]$
Return $dk[\Lambda]$

Figure 4: Games defining signature unforgeability under exposures of key-updatable digital signature scheme DS, and ciphertext indistinguishability under exposures of key-updatable public-key encryption scheme PKE.

Exp it can ask to be given the current signing key of the specified user. When it does so, $\vec{\Delta}'[\Lambda]$ is used to store the vector of strings the key was updated with.

At the end of the game the adversary outputs a user id Λ, a signature σ, a message m, and a key update vector $\vec{\Delta}$. The adversary has cheated if it previously received σ as the result of calling $\textsc{Sign}(\Lambda, m)$ and $\vec{\Delta} = \vec{\Delta}^*[\Lambda]$, or if it exposed the signing key of Λ and $\vec{\Delta}'[\Lambda]$ is a prefix

of $\vec{\Delta}$. It wins if it has not cheated and if σ verifies for m when the verification key of Λ is updated with $\vec{\Delta}$. The advantage of $\mathcal{A}_{\mathsf{DS}}$ in breaking the UFEXP security of DS is defined by $\mathsf{Adv}^{\mathsf{ufexp}}_{\mathsf{DS}}(\mathcal{A}_{\mathsf{DS}}) = \Pr[\mathrm{UFEXP}^{\mathcal{A}_{\mathsf{DS}}}_{\mathsf{DS}}]$.

Construction. In Appendix C we use a forward-secure [5] key-evolving signature scheme with unique signatures to construct a key-updatable signature scheme that is secure with respect to both of the above definitions. Roughly, a key-evolving signature scheme is like a key-updatable digital signature scheme that can only update with $\Delta = \varepsilon$. In order to enable updates with respect to arbitrary key update information, we sign each update string with the current key prior to evolving the key, and then include these intermediate signatures with our final signature.

3.2 Key-updatable public-key encryption schemes

We start by formally defining the syntax and correctness of a key-updatable public-key encryption. Then we specify a security definition for it. We will briefly sketch how to construct such a scheme, but leave the details to Appendix D. We consider public-key encryption with labels as introduced by Shoup [41].

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

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

The correctness condition requires that ciphertexts decrypt correctly as long as the encryption and decryption key are both updated with the same sequence of key update information $\vec{\Delta} = (\Delta_1, \Delta_2, \ldots)$. To formalize this, consider game PKECORR of Fig. 2, associated to a key-updatable public-key encryption scheme PKE and an adversary $\mathcal{C}$. The advantage of an adversary $\mathcal{C}$ against the correctness of PKE is given by $\mathsf{Adv}^{\mathsf{pkecorr}}_{\mathsf{PKE}}(\mathcal{C}) = \Pr[\mathrm{PKECORR}^{\mathcal{C}}_{\mathsf{PKE}}]$. Correctness requires that $\mathsf{Adv}^{\mathsf{pkecorr}}_{\mathsf{PKE}}(\mathcal{C}) = 0$ for all (even computationally unbounded) adversaries $\mathcal{C}$. See Section 4 for discussion on game-based definitions of correctness.

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

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

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

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

Ciphertext indistinguishability under exposures. Consider game INDEXP of Fig. 4, associated to a key-updatable public-key encryption scheme PKE and an adversary $\mathcal{A}_{\mathsf{PKE}}$. Roughly, it requires that PKE maintain CCA security [3] even if $\mathcal{A}_{\mathsf{PKE}}$ is given the decryption key (as long as that decryption key is no longer able to decrypt any challenge ciphertexts).

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

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

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

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

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

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

4 Bidirectional cryptographic channels

In this section we formally define the syntax and correctness of bidirectional cryptographic channels. Our notion of bidirectional channels will closely match that of Marson and Poettering [31]. Compared to their definition, we allow the receiving algorithm to be randomized and provide an alternative correctness condition. We argue that the new correctness condition is more appropriate for our desired use case of secure messaging. Henceforth, we will omit the adjective "bidirectional" and refer simply to channels.

Syntax of channel. A channel provides a method for two users to exchange messages in an arbitrary order. We will refer to the two users of a channel as the initiator $\mathcal{I}$ and the receiver $\mathcal{R}$. There will be no formal distinction between the two users, but when specifying attacks we follow the

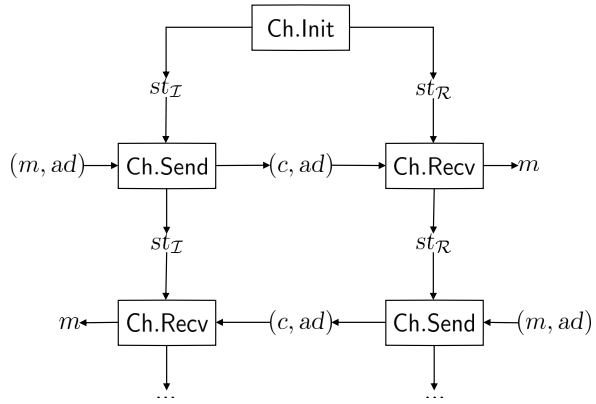

Figure 5: The interaction between bidirectional cryptographic channel algorithms.

---

convention of having $\mathcal{I}$ send a ciphertext first. We will use u as a variable to represent an arbitrary user and $\overline{u}$ to represent the other user. More formally, when $u \in \{\mathcal{I}, \mathcal{R}\}$ we let $\overline{u}$ denote the sole element of $\{\mathcal{I}, \mathcal{R}\} \setminus \{u\}$. Consider Fig. 5 for an overview of algorithms that constitute a channel Ch, and the interaction between them.

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

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

Correctness of channel. In Fig. 6 we provide two games, defining two alternative correctness requirements for a cryptographic channel. Lines labelled with the name of a game are included only in that game. The games differ in whether the adversary is given access to an oracle ROBUST or to an oracle REJECT. Game CORR uses the former, whereas game CORR$\perp$ uses the latter. The advantage of an adversary $\mathcal{C}$ against the correctness of channel Ch is given by $\mathsf{Adv}^{\mathsf{corr}}_{\mathsf{Ch}}(\mathcal{C}) = \Pr[\mathrm{CORR}^{\mathcal{C}}_{\mathsf{Ch}}]$ in one case, and $\mathsf{Adv}^{\mathsf{corr}^{\perp}}_{\mathsf{Ch}}(\mathcal{C}) = \Pr[\mathrm{CORR}{\perp}^{\mathcal{C}}_{\mathsf{Ch}}]$ in the other case. Correctness with respect to either notion requires that the advantage is equal 0 for all (even computationally unbounded) adversaries $\mathcal{C}$.

Our use of games to define correctness conditions follows the work of Marson and Poettering [31] and Bellare et. al. [9]. By considering unbounded adversaries and requiring an advantage of 0 we capture a typical information-theoretic perfect correctness requirement without having to explicitly quantify over sequences of actions. In this work we require only the perfect correctness because it is achieved by our scheme; however, it would be possible to capture computational correctness by

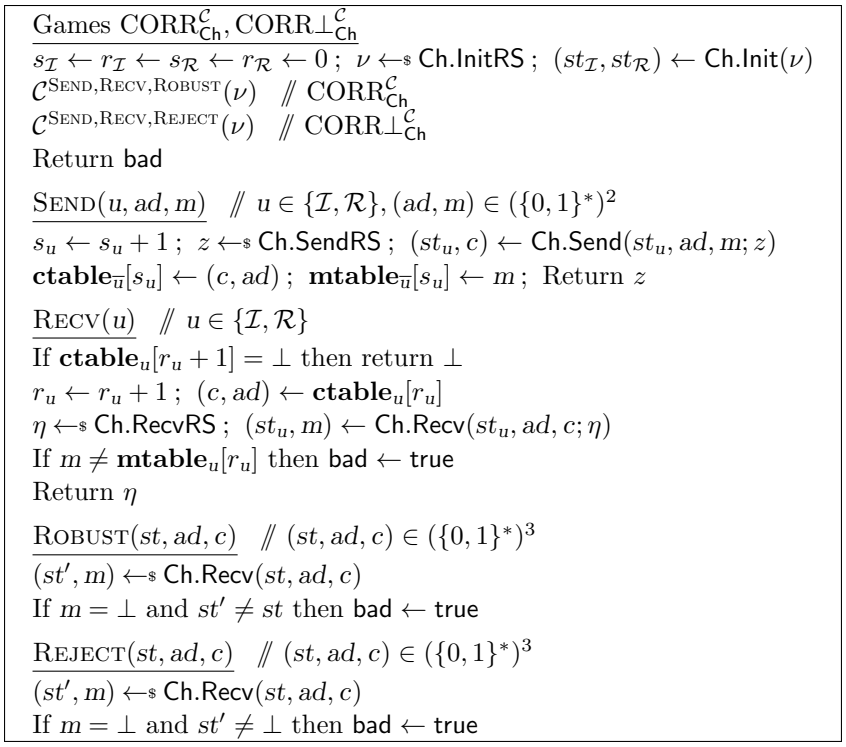

Figure 6: Games defining correctness of channel Ch. Lines labelled with the name of a game are included only in that game. CORR requires that Ch be robust when given an incorrect ciphertext via oracle Robust. CORR⊥ requires that Ch permanently returns ⊥ when given an incorrect ciphertext via oracle Reject.

considering a restricted class of adversaries.

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

The games differ in how the channel is required to behave in the case that a ciphertext is rejected. Game CORR (using oracle Robust) requires that the state of the user not be changed so that the channel can continue to be used. Game CORR⊥ (using oracle Reject) requires that the state of the user is set to ⊥. According to our conventions about the behavior of algorithms given ⊥ as input (see Section 2), the channel will then refuse to perform any further actions by setting all subsequent outputs to ⊥. We emphasize that the adversary specifies all inputs to Ch.Recv when making calls to Robust and Reject, so the behavior of those oracles is not related to the behavior of the other two oracles for which the game maintains the state of both users.

Comparison of correctness notions. The correctness required by CORR⊥ is identical to that of Marson and Poettering [31]. The CORR notion of correctness instead uses a form of robustness analogous to that of [9]. In Appendix B we discuss how these correctness notions have different implications for the *security* of the channel. It is trivial to convert a CORR-correct channel to a CORR⊥-correct channel and vice versa. Thus we will, without loss of generality, only provide a

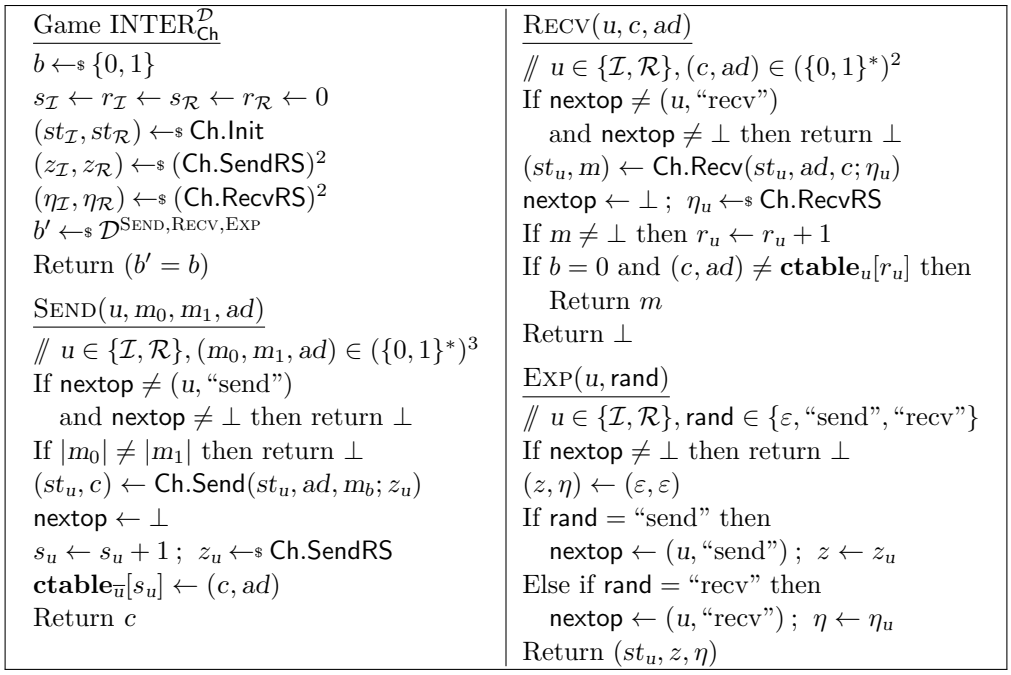

Figure 7: Game defining interface between adversary $\mathcal{D}$ and channel Ch.

scheme achieving CORR-correctness.

5 Security notion for channels

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

To introduce our security notion we will first describe a simple interface of how the adversary is allowed to interact with the channel. Then we show attacks that would break the security of *any* channel using this interface. Our final security notion will be created by adding checks to the interface that prevents adversary from performing any sequence of actions that leads to these unpreventable breaches of security. We introduce only the minimal necessary restrictions preventing the attacks, making sure that we allow *all* adversaries that do not trivially break the security as per above.

5.1 Channel interface game

Consider game INTER in Fig. 7. It defines the interface between an adversary $\mathcal{D}$ and a channel Ch. A secret bit b is chosen at random and the adversary's goal is to guess this bit given access to a left-or-right sending oracle, real-or-$\perp$ receiving oracle, and an exposure oracle. The sending oracle takes as input a user $u \in \{\mathcal{I}, \mathcal{R}\}$, two messages $m_0, m_1 \in \{0,1\}^*$, and associated data ad. Then it returns the encryption of m_b with ad by user u. The receiving oracle RECV takes as input

a user u, a ciphertext c, and associated data ad. It has user u decrypt this ciphertext using ad, and proceeds as follows. If $b = 0$ holds (along with another condition we discuss momentarily) then it returns the valid decryption of this ciphertext; otherwise it returns $\perp$. The exposure oracle EXP takes as input a user u, and a flag rand. It returns user's state st_u, and it might return random coins that will be used the next time this user runs algorithms $\mathsf{Ch.Send}$ or $\mathsf{Ch.Recv}$ (depending on the value of rand, which we discuss below). The advantage of adversary $\mathcal{D}$ against channel Ch is defined by $\mathsf{Adv}^{\mathsf{inter}}_{\mathsf{Ch}}(\mathcal{D}) = 2\Pr[\mathrm{INTER}^{\mathcal{D}}_{\mathsf{Ch}}] - 1$.

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

Let us consider the different cases of how a user's secrets might be exposed. They could be exposed while the user is in the middle of performing a $\mathsf{Ch.Send}$ operation, in the middle of performing a $\mathsf{Ch.Recv}$ operation, or when the user is idle (i.e. not in the middle of performing $\mathsf{Ch.Send}$ or $\mathsf{Ch.Recv}$). In the last case we expect the adversary to learn the user's state st_u, but nothing else. If the adversary is exposing the user during an operation, they would potentially learn the state before the operation, any secrets computed during the operation, and the state after the operation. We capture this by leaking the state from before the operation along with the randomness that will be used when the adversary makes its next query to SEND or RECV. This allows the adversary to compute the next state as well. The three possible values of rand are rand = "send" for the first possibility, rand = "recv" for the second possibility, and $\mathsf{rand} = \varepsilon$ for the third. These exposures represent what the adversary is learning while a particular operation is occurring, so we require (via nextop) that after such an exposure it immediately makes the corresponding oracle query. Without the use of the exposure oracle the game specified by this interface would essentially be equivalent to the combination of the integrity and confidentiality security notions defined by Marson and Poettering [31] in the all-in-one definition style of Rogaway and Shrimpton [38].

The interface game already includes some standard checks. First, we require that on any query (u, m_0, m_1, ad) to SEND the adversary must provide equal length messages. If the adversary does not do so (i.e. $|m_0| \neq |m_1|$) then SEND returns $\perp$ immediately. This prevents the inherent attack where an adversary could distinguish between the two values of b by asking for encryptions of different length messages and checking the length of the output ciphertext. Adversary $\mathcal{D}_1$ in Fig. 8 does just that and would achieve $\mathsf{Adv}^{\mathsf{inter}}_{\mathsf{Ch}}(\mathcal{D}_1) > 1/2$ against any channel Ch if not for that check.

Second, we want to prevent RECV from decrypting ciphertexts that are simply forwarded to it from SEND. So for each user u we keep track of counters s_u and r_u that track how many messages that user has sent and received. Then at the end of a SEND call to u the ciphertext-associated data pair (c, ad) is stored in the table $\mathbf{ctable}_{\overline{u}}$ with index s_u. When RECV is called for user $\overline{u}$ it will compare the pair (c, ad) against $\mathbf{ctable}_{\overline{u}}[r_{\overline{u}}]$ and if the pair matches return $\perp$ regardless of the value of the secret bit. If we did not do this check then for any channel Ch the adversary $\mathcal{D}_2$ shown in Fig. 8 would achieve $\mathsf{Adv}^{\mathsf{inter}}_{\mathsf{Ch}}(\mathcal{D}_2) = 1$.

We now specify several efficient adversaries that will have high advantage for *any* choice of Ch. For concreteness we always have our adversaries immediately start the actions required to perform the attacks, but all of the attacks would still work if the adversary had performed a number of unrelated procedure calls first. Associated data will never be important for our attacks so we will always set it to ε. We will typically set $m_0 = 0$ and $m_1 = 1$. For the following we let Ch be any channel and consider the adversaries shown in Fig. 8.

Adversary $\mathcal{D}_1^{\text{Send},\text{Recv},\text{Exp}}$
$(st, z, \eta) \leftarrow \text{Exp}(\mathcal{I}, \varepsilon)$
$n \leftarrow \max_{c \in [\mathsf{Ch.Send}(st,\varepsilon,1)]} |c|$
$m \leftarrow\!\!\$\ \{0,1\}^{n+2}$
$c \leftarrow \text{Send}(\mathcal{I}, m, 1, \varepsilon)$
If $|c| \le n$ then return 1
Return 0

Adversary $\mathcal{D}_2^{\text{Send},\text{Recv},\text{Exp}}$
$c \leftarrow \text{Send}(\mathcal{I}, 1, 1, \varepsilon)$
$m \leftarrow \text{Recv}(\mathcal{R}, c, \varepsilon)$
If $m = \perp$ then return 1
Return 0

Adversary $\mathcal{D}_3^{\text{Send},\text{Recv},\text{Exp}}$
$(st, z, \eta) \leftarrow \text{Exp}(\mathcal{I}, \varepsilon)$
$(st, c) \leftarrow\!\!\$\ \mathsf{Ch.Send}(st, \varepsilon, 1)$
$m \leftarrow \text{Recv}(\mathcal{R}, c, \varepsilon)$
If $m = \perp$ then return 1
Return 0

Adversary $\mathcal{D}_{3.1}^{\text{Send},\text{Recv},\text{Exp}}$
$(st, z, \eta) \leftarrow \text{Exp}(\mathcal{I}, \varepsilon)$
$(st, c) \leftarrow\!\!\$\ \mathsf{Ch.Send}(st, \varepsilon, 1)$
$m \leftarrow \text{Recv}(\mathcal{R}, c, \varepsilon)$
$(st, c) \leftarrow\!\!\$\ \mathsf{Ch.Send}(st, \varepsilon, 1)$
$m \leftarrow \text{Recv}(\mathcal{R}, c, \varepsilon)$
If $m = \perp$ then return 1
Return 0

Adversary $\mathcal{D}_{3.2}^{\text{Send},\text{Recv},\text{Exp}}$
$(st, z, \eta) \leftarrow \text{Exp}(\mathcal{I}, \varepsilon)$
$(st, c) \leftarrow\!\!\$\ \mathsf{Ch.Send}(st, \varepsilon, 1)$
$m \leftarrow \text{Recv}(\mathcal{R}, c, \varepsilon)$
$c \leftarrow \text{Send}(\mathcal{R}, 0, 1, \varepsilon)$
$(st, m) \leftarrow\!\!\$\ \mathsf{Ch.Recv}(st, \varepsilon, c)$
If $m = 1$ then return 1
Return 0

Adversary $\mathcal{D}_4^{\text{Send},\text{Recv},\text{Exp}}$
$c \leftarrow \text{Send}(\mathcal{I}, 0, 1, \varepsilon)$
$(st, z, \eta) \leftarrow \text{Exp}(\mathcal{R}, \varepsilon)$
$(st, m) \leftarrow\!\!\$\ \mathsf{Ch.Recv}(st, \varepsilon, c)$
If $m = 1$ then return 1
Return 0

Adversary $\mathcal{D}_5^{\text{Send},\text{Recv},\text{Exp}}$
$(st, z, \eta) \leftarrow \text{Exp}(\mathcal{R}, \varepsilon)$
$c \leftarrow \text{Send}(\mathcal{I}, 0, 1, \varepsilon)$
$(st, m) \leftarrow\!\!\$\ \mathsf{Ch.Recv}(st, \varepsilon, c)$
If $m = 1$ then return 1
Return 0

Adversary $\mathcal{D}_6^{\text{Send},\text{Recv},\text{Exp}}$
$(st, z, \eta) \leftarrow \text{Exp}(\mathcal{I}, \text{“send”})$
$(st, c) \leftarrow \mathsf{Ch.Send}(st, \varepsilon, 1; z)$
$c' \leftarrow \text{Send}(\mathcal{I}, 0, 1, \varepsilon)$
If $c' = c$ then return 1
Return 0

Figure 8: Generic attacks against any channel Ch with interface INTER.

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

After an adversary has done the above to trivially send a forgery to $\overline{u}$ it can easily perform further attacks on both the integrity and authenticity of the channel. These are shown by adversaries $\mathcal{D}_{3.1}$ and $\mathcal{D}_{3.2}$. The first displays the fact that the attacker can easily send further forgeries to $\overline{u}$. The second displays the fact that the attacker can now easily decrypt any messages sent by $\overline{u}$. We have $\mathsf{Adv}^{\mathsf{inter}}_{\mathsf{Ch}}(\mathcal{D}_{3.1}) = 1$ and $\mathsf{Adv}^{\mathsf{inter}}_{\mathsf{Ch}}(\mathcal{D}_{3.2}) = 1$.

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

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

Exposing Randomness. If an adversary exposes user u with rand = “send” then it is able to compute the next state of u by running $\mathsf{Ch.Send}$ locally with the same randomness that u will use. So in this case the security game must act as if the adversary exposed both the current and the next state. In particular, the attacks above could only succeed until, first, the exposed user u updated its secrets

and, second, user $\overline{u}$ updates its secrets accordingly (which can happen after it receives the next message from u). But if the randomness was exposed, then secrets would need to be updated at least twice until the security is restored.

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

Exposing with rand = "recv" does not generically endow the adversary with the ability to do any additional attacks.

5.2 Optimal security of a channel

Our full security game is obtained by adding a minimal amount of code to INTER to disallow the generic attacks just discussed. Consider the game AEAC (authenticated encryption against compromise) shown in Fig. 9. We define the advantage of an adversary $\mathcal{D}$ against channel Ch by $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{Ch}}(\mathcal{D}) = 2\Pr[\mathrm{AEAC}^{\mathcal{D}}_{\mathsf{Ch}}] - 1$.

We now have a total of eight variables to control the behavior of the adversary and prevent it from abusing trivial attacks. Some of the variables are summarized in Fig. 10. We have already seen s_u, r_u, nextop, and $\mathbf{ctable}_u$ in INTER. The new variables we have added in AEAC are tables $\mathbf{forge}_u$ and $\mathbf{ch}_u$, number $\mathcal{X}_u \in \mathbb{N}$, and flag $\mathsf{restricted}_u \in \{\mathsf{true}, \mathsf{false}\}$. We now discuss the new variables.

The table $\mathbf{forge}_u$ was added to prevent the type of attack shown in $\mathcal{D}_3$. When the adversary calls EXP on user u we set $\mathbf{forge}_{\overline{u}}$ to "trivial" for the indices of ciphertexts for which this adversary is now necessarily able to create forgeries. If the adversary takes advantage of this to send a ciphertext of its own creation to $\overline{u}$ then the flag $\mathsf{restricted}_{\overline{u}}$ will be set, whose effect we will describe momentarily.

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

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

Finally the flag $\mathsf{restricted}_u$ serves to both allow and disallow some attacks. The flag is initialized to false. It is set to true when the adversary forges a ciphertext to u after exposing $\overline{u}$. Once u has received a different ciphertext than was sent by $\overline{u}$ there is no reason to think that u should be able to decrypt ciphertexts sent by $\overline{u}$ or send its own ciphertexts to $\overline{u}$. As such, if u is restricted (i.e.

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

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

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

$b \leftarrow\!\!\$\ \{0,1\}$; $s_{\mathcal{I}} \leftarrow 0$; $r_{\mathcal{I}} \leftarrow 0$; $s_{\mathcal{R}} \leftarrow 0$; $r_{\mathcal{R}} \leftarrow 0$
$\mathsf{restricted}_{\mathcal{I}} \leftarrow \mathsf{false}$; $\mathsf{restricted}_{\mathcal{R}} \leftarrow \mathsf{false}$
$\mathbf{forge}_{\mathcal{I}}[0 \ldots \infty] \leftarrow$ "nontrivial" ; $\mathbf{forge}_{\mathcal{R}}[0 \ldots \infty] \leftarrow$ "nontrivial"
$\mathcal{X}_{\mathcal{I}} \leftarrow 0$; $\mathcal{X}_{\mathcal{R}} \leftarrow 0$; $(st_{\mathcal{I}}, st_{\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{Ch.Init}$
$(z_{\mathcal{I}}, z_{\mathcal{R}}) \leftarrow\!\!\$\ (\mathsf{Ch.SendRS})^2$; $(\eta_{\mathcal{I}}, \eta_{\mathcal{R}}) \leftarrow\!\!\$\ (\mathsf{Ch.RecvRS})^2$
$b' \leftarrow\!\!\$\ \mathcal{D}^{\text{SEND,RECV,EXP}}$
Return $(b' = b)$

$\text{SEND}(u, m_0, m_1, ad)$ // $u \in \{\mathcal{I}, \mathcal{R}\}, (m_0, m_1, ad) \in (\{0,1\}^*)^3$
If $\mathsf{nextop} \neq (u, \text{"send"})$ and $\mathsf{nextop} \neq \perp$ then return $\perp$
If $|m_0| \neq |m_1|$ then return $\perp$
If ($r_u < \mathcal{X}_u$ or $\mathsf{restricted}_u$ or $\mathbf{ch}_u[s_u + 1] =$ "forbidden") and $m_0 \neq m_1$ then
  Return $\perp$
$(st_u, c) \leftarrow \mathsf{Ch.Send}(st_u, ad, m_b; z_u)$
$\mathsf{nextop} \leftarrow \perp$; $s_u \leftarrow s_u + 1$; $z_u \leftarrow\!\!\$\ \mathsf{Ch.SendRS}$
If not $\mathsf{restricted}_u$ then $\mathbf{ctable}_{\overline{u}}[s_u] \leftarrow (c, ad)$
If $m_0 \neq m_1$ then $\mathbf{ch}_u[s_u] \leftarrow$ "done"
Return c

$\text{RECV}(u, c, ad)$ // $u \in \{\mathcal{I}, \mathcal{R}\}, (c, ad) \in (\{0,1\}^*)^2$
If $\mathsf{nextop} \neq (u, \text{"recv"})$ and $\mathsf{nextop} \neq \perp$ then return $\perp$
$(st_u, m) \leftarrow \mathsf{Ch.Recv}(st_u, ad, c; \eta_u)$
$\mathsf{nextop} \leftarrow \perp$; $\eta_u \leftarrow\!\!\$\ \mathsf{Ch.RecvRS}$
If $m = \perp$ then return $\perp$
$r_u \leftarrow r_u + 1$
If $\mathbf{forge}_u[r_u] =$ "trivial" and $(c, ad) \neq \mathbf{ctable}_u[r_u]$ then
  $\mathsf{restricted}_u \leftarrow \mathsf{true}$
If $\mathsf{restricted}_u$ or ($b = 0$ and $(c, ad) \neq \mathbf{ctable}_u[r_u]$) then
  Return m
Return $\perp$

$\text{EXP}(u, \mathsf{rand})$ // $u \in \{\mathcal{I}, \mathcal{R}\}, \mathsf{rand} \in \{\varepsilon, \text{"send"}, \text{"recv"}\}$
If $\mathsf{nextop} \neq \perp$ then return $\perp$
If $\mathsf{restricted}_u$ then return (st_u, z_u, η_u)
If $\exists i \in (r_u, s_{\overline{u}}]$ s.t. $\mathbf{ch}_{\overline{u}}[i] =$ "done" then
  Return $\perp$
$\mathbf{forge}_{\overline{u}}[s_u + 1] \leftarrow$ "trivial" ; $(z, \eta) \leftarrow (\varepsilon, \varepsilon)$; $\mathcal{X}_{\overline{u}} \leftarrow s_u + 1$
If $\mathsf{rand} =$ "send" then
  $\mathsf{nextop} \leftarrow (u, \text{"send"})$; $z \leftarrow z_u$; $\mathcal{X}_{\overline{u}} \leftarrow s_u + 2$
  $\mathbf{forge}_{\overline{u}}[s_u + 2] \leftarrow$ "trivial" ; $\mathbf{ch}_u[s_u + 1] \leftarrow$ "forbidden"
Else if $\mathsf{rand} =$ "recv" then
  $\mathsf{nextop} \leftarrow (u, \text{"recv"})$; $\eta \leftarrow \eta_u$
Return (st_u, z, η)

Figure 9: Game defining AEAC security of channel Ch.

$\mathsf{restricted}_u = \mathsf{true}$) we will not add its ciphertexts to $\mathbf{ctable}_{\overline{u}}$, we will always show the true output when u attempts to decrypt ciphertexts given to it by the adversary (even if they were sent by $\overline{u}$), and if the adversary asks to expose u we will return all of its secret state without setting any of the other variables that would restrict the actions the adversary is allowed to take.

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

Figure 10: Table summarizing some important variables in game AEAC. A "$-$" indicates a way in which the behavior of the adversary is being restricted. A "$+$" indicates a way in which the behavior of the adversary is being enabled.

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

Informal description of the security game. In Section 5.3 we provide a thorough written description of our security model to facilitate high-level understanding of it. For intricate security definitions like ours there is often ambiguity or inconsistency in subtle corner cases of the definition when written out fully in text. As such this description should merely be considered an informal aid while the pseudocode of Fig. 9 is the actual definition.

Comparison to recent definitions. The three recents works we studied while deciding how to write our security definition were [16], [9], and [28]. Their settings were all distinct, but each presented security models that involve different "stages" of keys. All three works made distinct decisions in how to address challenges in different stages. In Appendix A we discuss these decisions, noting that they result in qualitatively identical but quantitatively distinct definitions.

5.3 Informal description of our security definition

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

We first describe sending of messages and receiving of ciphertexts, without reference to the restrictions that will be placed on these operations after the exposure of a user's secrets. When the adversary asks user u to send a message, it will provide two messages m_0 and m_1 of equal length,

together with associated data ad. Then algorithm $\mathsf{Ch.Send}$ will be run on input the state of u, m_b, and ad. The resulting ciphertext will be returned to the adversary. When the adversary asks user u to receive a ciphertext, it will provide the ciphertext c and associated data ad. Then algorithm $\mathsf{Ch.Recv}$ will be run on input the state of u, c, and ad. The adversary gets back $\perp$ if the decryption of c with ad helps adversary to trivially win the game (i.e. it is identical to the ciphertext sent by $\overline{u}$, and only if the last ciphertext sent by $\overline{u}$ was already accepted by u). If the decryption of the provided ciphertext failed, the adversary gets $\perp$ as the result. Otherwise the actual decryption will be returned if $b = 0$, and $\perp$ will be returned if $b = 1$.

We proceed to describe the effect of exposing a user, which introduces the majority of the complexity of our model. When exposing the current state of user u the adversary will provide an additional string rand that can equal ε, "send", or "recv". The flag rand indicates whether the adversary is exposing the state of the user while they are "at rest", while they are sending a message, or while they are receiving a ciphertext. In the latter two cases the adversary is required to follow up with a valid request that the specified user performs the specified action, and the adversary will be proactively given back the randomness that will be used when this action is performed. Additionally, the expose query to u is not allowed if it allows the adversary to trivially win the game, which happens if $\overline{u}$ was asked to send a message to u (encrypting one of two challenge plaintexts) that was not yet received by u.

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

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

When a user u is restricted a number of the statements we made above no longer hold. When a restricted user is exposed, their current state, next randomness for sending, and next randomness for receiving will be freely given to the adversary without placing any restrictions on its future actions. Similarly, there is no longer any restriction on when u can be exposed (except if $\overline{u}$ is not restricted then u cannot be exposed after $\overline{u}$ was exposed while performing an operation, until $\overline{u}$ has finished performing the corresponding operation). The adversary is no longer allowed to ask u to send one of two different messages. When u is asked to receive a ciphertext, the adversary will only be given back $\perp$ if the decryption failed, meaning the adversary always gets back the actual output of $\mathsf{Ch.Recv}$ (no other conditions will trigger $\perp$ to be returned). Finally, a query asking $\overline{u}$ to accept a ciphertext no longer always returns $\perp$ if this ciphertext was produced by querying a restricted user u to send a message. Instead, if the previous message sent from u was accepted by $\overline{u}$, then the above query returns the actual output of $\mathsf{Ch.Recv}$ if $b = 0$, and $\perp$ otherwise.

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

but again refer the reader to our pseudocode for the precise definition.

6 Construction of a secure channel

6.1 Our construction

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

In this section we construct our cryptographic channel and motivate our design decisions by giving attacks against variants of the channel. In Section 6.2 we will prove its security by reducing it to that of its underlying components.

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

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

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

Cryptographic channel $\mathsf{SCH}[\mathsf{DS}, \mathsf{PKE}, \mathsf{H}]$. Let DS be a key-updatable digital signature scheme, PKE be a key-updatable public-key encryption scheme, and H be a family of functions. We build a cryptographic channel $\mathsf{SCh} = \mathsf{SCH}[\mathsf{DS}, \mathsf{PKE}, \mathsf{H}]$ as defined in Fig. 11.

A user's state st_u, among other values, contains counters s_u, r_u, r_u^{ack}. Here, s_u is the number of messages that u sent to $\overline{u}$, and r_u is the number of messages they received back from $\overline{u}$. The counter r_u^{ack} stores the last value of $r_{\overline{u}}$ in a ciphertext received by u (i.e. the index of the last ciphertext that u believes $\overline{u}$ has received and acknowledged). This counter is used to ensure that prior to running a signature verification algorithm, the vertification key vk is updated with respect to the same transcripts as the signing key sk (at the time it was used to produce the signature). Note that algorithm $\mathsf{DS.Vrfy}$ returns (vk'', t) where t is the result of verifying that σ is a valid signature for v with respect to verification key vk'' (using the notation convention from Section 3).

Algorithm $\mathsf{SCh.Init}$

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

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

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

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

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

Figure 11: Construction of channel $\mathsf{SCh} = \mathsf{SCH}[\mathsf{DS}, \mathsf{PKE}, \mathsf{H}]$ from function family H, key-updatable digital signature scheme DS, and key-updatable public-key encryption scheme PKE.

Inefficiencies of SCh. A few aspects of SCh are less efficient than one would a priori hope. The state maintained by a user u (specifically the tables $\mathbf{dk}_u$ and $\boldsymbol{\tau}_{s,u}$) is not constant in size, but instead grows linearly with the number of ciphertexts that u sent to $\overline{u}$ without receiving a reply back. Additionally, when DS is instantiated with the particular choice of DS that we define in Appendix C the length of the ciphertext sent by a user u and the amount of state it stores will grow linearly in the number of ciphertexts that u has received since the last time they sent a ciphertext. Such inefficiencies would be unacceptable for a protocol like TLS or SSH, but in our motivating context of messaging is it plausible that they are acceptable. Each message is human generated and the state gets "refreshed" regularly if the two users regularly reply to one another. One could additionally consider designing an app to regularly send an empty message whose sole purpose is state refreshing. We leave as interesting future work improving on the efficiency of our construction.

Design decisions. We will now discuss attacks against different variants of SCh. This serves to motivate the decisions made in its design and give intuition for why it achieves the desired security.

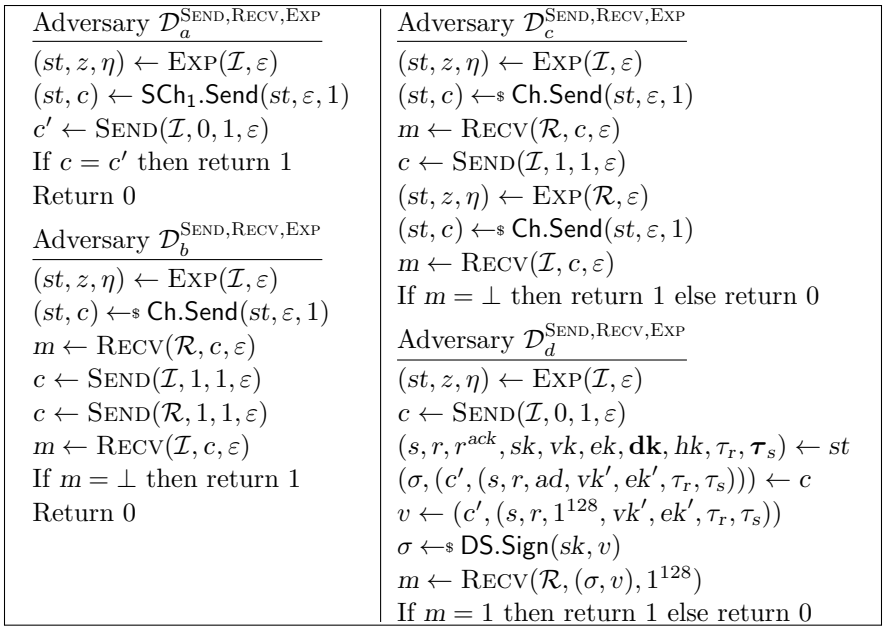

Figure 12: Attacks against variants of SCh.

Several steps in the security proof of this construction can be understood by noting which of these attacks are ruled out in the process.

The attacks are shown in Fig. 12 and Fig. 13. The first several attacks serve to demonstrate that Ch.Send must use a sufficient amount of randomness (shown in $\mathcal{D}_a$, $\mathcal{D}_b$, $\mathcal{D}_c$) and that H needs to be collision resistant (shown in $\mathcal{D}_b$, $\mathcal{D}_c$). The next attack shows why our construction would be insecure if we did not use labels with PKE (shown in $\mathcal{D}_d$). Then we provide two attacks showing why the keys of DS and PKE need to be updated (shown in $\mathcal{D}_e$, $\mathcal{D}_f$). Then we show an attack that arises if multiple valid signatures can be found for the same string (shown in $\mathcal{D}_g$). Finally, we conclude with attacks that would apply if we used symmetric instead of asymmetric primitives to build SCh (shown in $\mathcal{D}_h$, $\mathcal{D}_i$).

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

Consider the adversary $\mathcal{D}_a$. It exposes $\mathcal{I}$, encrypts the message 1 locally, and then sends a challenge query to $\mathcal{I}$ asking for the encryption of either 1 or 0. By comparing the ciphertext it produced to the one returned by SEND it can determine which message was encrypted, learning the secret bit. We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_1}(\mathcal{D}_a) = 1$. This attack is fairly straightforward and will be ruled out by the security of PKE in our proof without having to be addressed directly.

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

same ciphertext that $\mathcal{I}$ sent. Different attacks are possible from this point.

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

Scheme without collision-resistant hashing. If it is easy to find collisions in H then we can attack the channel by causing both parties to have matching transcripts despite having seen different sequences of ciphertexts. For concreteness let SCh_2 be a variant of our scheme using a hash function that outputs 0^{128} on all inputs. Let $\mathsf{Ch} = \mathsf{SCh}_2$ and again consider adversaries $\mathcal{D}_b$ and $\mathcal{D}_c$. We no longer expect the ciphertexts that they produce locally to match the ciphertexts returned by $\mathcal{I}$. However, they will have the same hash value and thus produce the same transcript $\tau_{r,\mathcal{R}} = 0^{128} = \tau_{s,\mathcal{I}}$. Consequently, $\mathcal{R}$ still updates its signing key in the same way regardless of whether it receives the ciphertext produced by $\mathcal{I}$ or the ciphertext locally generated by adversary. So the messages subsequently sent by $\mathcal{R}$ will still be accepted by $\mathcal{I}$. We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_2}(\mathcal{D}_b) = 1$ and $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_2}(\mathcal{D}_c) = 1$.

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

Schemes without updatable keys. We will now show why it is necessary to define new forms of PKE and DS for our construction.

Let SCh_4 be a variant of SCh that uses a digital signature scheme that does not update its keys. Consider adversary $\mathcal{D}_e$. It exposes $\mathcal{I}$, then queries SEND for $\mathcal{I}$ to send a message to $\mathcal{R}$, but uses the exposed secrets to replace it with a locally produced ciphertext c. It calls RECV for $\mathcal{R}$ with c, which sets $\mathsf{restricted}_{\mathcal{R}} = \mathsf{true}$. Since the signing key is not updated in SCh_4, the adversary now exposes $\mathcal{R}$ to obtain a signing key whose signatures will be accepted by $\mathcal{I}$. It uses this to forge a ciphertext to $\mathcal{I}$ to learn the secret bit. We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_4}(\mathcal{D}_e) = 1$.

Let SCh_5 be a variant of SCh that uses a public-key encryption scheme that does not update its keys. Consider adversary $\mathcal{D}_f$. It exposes $\mathcal{I}$ and uses this to send $\mathcal{R}$ a different ciphertext than is sent by $\mathcal{I}$ (setting $\mathsf{restricted}_{\mathcal{R}} = \mathsf{true}$). Since the decryption key is not updated, the adversary now exposes $\mathcal{R}$ to obtain a decryption key that can be used to decrypt a challenge ciphertext sent by $\mathcal{I}$. We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_5}(\mathcal{D}_f) = 1$.

Scheme with non-unique signatures. Let SCh_6 be a variant of our scheme using a digital signature scheme that does not have unique signatures. For concreteness, assume that $\sigma \,\|\, sk$ is a valid signature whenever σ is. Then consider adversary $\mathcal{D}_g$. It exposes $\mathcal{I}$ and has $\mathcal{I}$ send a challenge ciphertext. Then it modifies the ciphertext by changing the signature and forwards this modified ciphertext on to $\mathcal{R}$. The adversary is given back the true decryption of this ciphertext (because

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

$(st, z, \eta) \leftarrow \text{EXP}(\mathcal{I}, \varepsilon)$
$c_{\mathcal{I}} \leftarrow \text{SEND}(\mathcal{I}, 1, 1, \varepsilon)$
$(\sigma, (c', (s, r, ad, vk_{\mathcal{I}}, ek_{\mathcal{I}}, \tau_r, \tau_s))) \leftarrow c_{\mathcal{I}}$
$(st, c) \leftarrow\!\!\$\ \mathsf{SCh_4.Send}(st, \varepsilon, 0)$
$m \leftarrow \text{RECV}(\mathcal{R}, c, \varepsilon)$
$(st, z, \eta) \leftarrow \text{EXP}(\mathcal{R}, \varepsilon)$
$(s, r, r^{ack}, sk, vk, ek, \mathbf{dk}, hk, \tau_r, \boldsymbol{\tau}_s) \leftarrow st$
$\tau_r \leftarrow \mathsf{H.Ev}(hk, c_{\mathcal{I}})$
$st \leftarrow (s, r, r^{ack}, sk, vk_{\mathcal{I}}, ek_{\mathcal{I}}, \mathbf{dk}, hk, \tau_r, \boldsymbol{\tau}_s)$
$(st, c) \leftarrow\!\!\$\ \mathsf{SCh_4.Send}(st, \varepsilon, 1)$
$m \leftarrow \text{RECV}(\mathcal{I}, c, \varepsilon)$
If $m = \perp$ then return 1 else return 0

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

$(st, z, \eta) \leftarrow \text{EXP}(\mathcal{I}, \varepsilon)$
$(\sigma, (c', \ell)) \leftarrow \text{SEND}(\mathcal{I}, 0, 1, \varepsilon)$
$(st, c) \leftarrow\!\!\$\ \mathsf{SCh_5.Send}(st, \varepsilon, 0)$
$m \leftarrow \text{RECV}(\mathcal{R}, c, \varepsilon)$
$(st, z, \eta) \leftarrow \text{EXP}(\mathcal{R}, \varepsilon)$
$(s, r, r^{ack}, sk, vk, ek, \mathbf{dk}, hk, \tau_r, \boldsymbol{\tau}_s) \leftarrow st$
$m \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}[0], \ell, c')$
If $m = 1$ then return 1 else return 0

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

$(st, z, \eta) \leftarrow \text{EXP}(\mathcal{I}, \varepsilon)$
$(\sigma, v) \leftarrow \text{SEND}(\mathcal{I}, 0, 1, \varepsilon)$
$(s, r, r^{ack}, sk, vk, ek, \mathbf{dk}, hk, \tau_r, \boldsymbol{\tau}_s) \leftarrow st$
$m \leftarrow \text{RECV}(\mathcal{R}, (\sigma \,\|\, sk, v), \varepsilon)$
If $m = 1$ then return 1
Return 0

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

$(st, z, \eta) \leftarrow \text{EXP}(\mathcal{I}, \varepsilon)$
$(s, r, r^{ack}, sk, vk, ek, \mathbf{dk}, hk, \tau_r, \boldsymbol{\tau}_s) \leftarrow st$
$st \leftarrow (s, r, r^{ack}, vk, sk, ek, \mathbf{dk}, hk, \tau_r, \boldsymbol{\tau}_s)$
$(st, c) \leftarrow\!\!\$\ \mathsf{SCh_7.Send}(st, \varepsilon, 0)$
$m \leftarrow \text{RECV}(\mathcal{I}, c, \varepsilon)$
If $m = \perp$ then return 1
Return 0

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

$(st, z, \eta) \leftarrow \text{EXP}(\mathcal{I}, \varepsilon)$
$(\sigma, (c', \ell)) \leftarrow \text{SEND}(\mathcal{I}, 0, 1, \varepsilon)$
$(s, r, r^{ack}, sk, vk, ek, \mathbf{dk}, hk, \tau_r, \boldsymbol{\tau}_s) \leftarrow st$
$m \leftarrow \mathsf{PKE.Dec}(ek, \ell, c')$
If $m = 1$ then return 1
Return 0

Figure 13: Attacks against variants of SCh.

it was changed) which trivially reveals the secret bit of the game (here it is important that the signature is not part of the label used for encryption/decryption). We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_6}(\mathcal{D}_g) = 1$.

Scheme with symmetric primitives. Let SCh_7 be a variant of our scheme that uses a MAC instead of a digital signature scheme (e.g. $vk = sk$ always, and vk is presumably no longer sent in the clear with the ciphertext). Consider adversary $\mathcal{D}_h$. It simply exposes $\mathcal{I}$ and then uses $\mathcal{I}$'s vk to send a message to $\mathcal{I}$. This trivially allows it to determine the secret bit. Here we used that PKE will decrypt any ciphertext to a non-$\perp$ value. We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_7}(\mathcal{D}_h) = 1$.

Similarly let SCh_8 be a variant of our scheme that uses symmetric encryption instead of public-key encryption (e.g. $ek = dk$ always, and ek is presumably no longer sent in the clear with the ciphertext). Adversary $\mathcal{D}_i$ exposes user $\mathcal{I}$ and then uses the corresponding ek to decrypt a challenge message encrypted by $\mathcal{I}$. We have $\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}_8}(\mathcal{D}_i) = 1$.

Stated broadly, a scheme that relies on symmetric primitives will not be secure because a user will know sufficient information to send a ciphertext that they would themselves accept or to read a message that they sent to the other user. Our security notion requires that this is not possible.

6.2 Security proof

The following theorem bounds the advantage of an adversary breaking the AEAC security of SCh using the advantages of adversaries against the CR security of H, the UFEXP and UNIQ security of DS, the INDEXP security of PKE, and the min-entropy of DS and PKE.

Theorem 6.1 *Let* DS *be a key-updatable digital signature scheme,* PKE *be a key-updatable public-key encryption scheme, and* H *be a family of functions. Let* $\mathsf{SCh} = \mathsf{SCH}[\mathsf{DS}, \mathsf{PKE}, \mathsf{H}]$. *Let* $\mathcal{D}$ *be an adversary making at most* q_{SEND} *queries to its* SEND *oracle,* q_{RECV} *queries to its* RECV *oracle, and* q_{EXP} *queries to its* EXP *oracle. Then we can build adversaries* $\mathcal{A}_{\mathsf{H}}$, $\mathcal{A}_{\mathsf{DS}}$, $\mathcal{B}_{\mathsf{DS}}$, *and* $\mathcal{A}_{\mathsf{PKE}}$ *such that*

$$\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}}(\mathcal{D}) \leq 2 \cdot (q_{\text{SEND}} \cdot 2^{-\mu} + \mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathcal{A}_{\mathsf{H}}) + \mathsf{Adv}^{\mathsf{ufexp}}_{\mathsf{DS}}(\mathcal{A}_{\mathsf{DS}}) + \mathsf{Adv}^{\mathsf{uniq}}_{\mathsf{DS}}(\mathcal{B}_{\mathsf{DS}})) + \mathsf{Adv}^{\mathsf{indexp}}_{\mathsf{PKE}}(\mathcal{A}_{\mathsf{PKE}}) \quad (1)$$

where $\mu = \mathrm{H}_\infty(\mathsf{DS.Kg}) + \mathrm{H}_\infty(\mathsf{PKE.Kg}) + \mathrm{H}_\infty(\mathsf{PKE.Enc})$. *Adversary* $\mathcal{A}_{\mathsf{DS}}$ *makes at most* $q_{\text{SEND}} + 2$ *queries to its* NEWUSER *oracle,* q_{SEND} *queries to its* SIGN *oracle, and* q_{EXP} *queries to its* EXP *oracle. Adversary* $\mathcal{B}_{\mathsf{DS}}$ *makes at most* $q_{\text{SEND}} + 2$ *queries to its* NEWUSER *oracle. Adversary* $\mathcal{A}_{\mathsf{PKE}}$ *makes at most* $q_{\text{SEND}} + 2$ *queries to its* NEWUSER *oracle,* $q_{\text{SEND}} * (q_{\text{RECV}} + 1)$ *queries to its* UPDEK *oracle,* $(q_{\text{SEND}} + 1) * q_{\text{RECV}}$ *queries to its* UPDDK *oracle,* q_{SEND} *queries to its* ENC *oracle,* q_{RECV} *queries to its* DEC *oracle, and* $q_{\text{SEND}} + 2$ *queries to its* EXP *oracle. Adversaries* $\mathcal{A}_{\mathsf{H}}$, $\mathcal{A}_{\mathsf{DS}}$, $\mathcal{B}_{\mathsf{DS}}$, *and* $\mathcal{A}_{\mathsf{PKE}}$ *all have runtime about that of* $\mathcal{D}$.

The proof broadly consists of two stages. The first stage of the proof (consisting of three game transitions) argues that the adversary will not be able to forge a ciphertext to an unrestricted user except by exposing the other user. This argument is justified by a reduction to an adversary $\mathcal{A}_{\mathsf{DS}}$ against the security of the digital signature scheme. However, care must be taken in this reduction to ensure that $\mathcal{D}$ cannot induce behavior in $\mathcal{A}_{\mathsf{DS}}$ that would result in $\mathcal{A}_{\mathsf{DS}}$ cheating in the digital signature game. Addressing this possibility involves arguing that $\mathcal{D}$ cannot predict any output of SEND (from whence the min-entropy term in the bound arises) and that it cannot find any collisions in the hash function H.

Once this stage is complete the output of RECV no longer depends on the secret bit b, so we move to using the security of PKE to argue that $\mathcal{D}$ cannot use SEND to learn the value of the secret bit. This is the second stage of the proof. But prior to this reduction we have to make one last argument using the security of DS. Specifically we show that, given a ciphertext (σ, v), the adversary will not be able to find a new signature σ' such that (σ', v) will be accepted by the receiver (otherwise since $\sigma \neq \sigma'$, oracle RECV would return the true decryption of this ciphertext which would be the same as the decryption of the original ciphertext and thus allow a trivial attack). Having done this, the reduction to the security of PKE is follows.

Proof of Theorem 6.1: For the proof we use games G_0, G_1, G_2, G_3 defined in Fig. 14, games G_4, G_5 defined in Fig. 17, and the adversaries defined in Fig. 15, 16, 18, 19.

The theorem follows immediately from the following seven claims. The first four claims correspond to the first stage of the proof as discussed in Section 6.2. The rest correspond to the second stage. For compactness we will let $q = q_{\text{SEND}}$ throughout the proof.

1. $\Pr[\mathrm{G}_0] = \Pr[\mathrm{AEAC}^{\mathcal{D}}_{\mathsf{SCh}}]$
2. $\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_1] \leq q \cdot 2^{-\mu}$
3. $\Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_2] \leq \mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathcal{A}_{\mathsf{H}})$
4. $\Pr[\mathrm{G}_2] - \Pr[\mathrm{G}_3] \leq \mathsf{Adv}^{\mathsf{ufexp}}_{\mathsf{DS}}(\mathcal{A}_{\mathsf{DS}})$
5. $\Pr[\mathrm{G}_3] = \Pr[\mathrm{G}_4]$
6. $\Pr[\mathrm{G}_4] - \Pr[\mathrm{G}_5] \leq \mathsf{Adv}^{\mathsf{uniq}}_{\mathsf{DS}}(\mathcal{B}_{\mathsf{DS}})$
7. $\Pr[\mathrm{G}_5] \leq \Pr[\mathrm{INDEXP}^{\mathcal{A}_{\mathsf{PKE}}}_{\mathsf{PKE}}]$

Referring to the adversaries from Section 6.1 we can roughly think as follows. Claim 1 and Claim 2 will rule out attacks like $\mathcal{D}_a$, $\mathcal{D}_b$, and $\mathcal{D}_c$. Claim 4 will rule out attacks like $\mathcal{D}_e$ and $\mathcal{D}_h$. Claim 6 will rule out attacks like $\mathcal{D}_g$. Claim 7 will rule out attacks like $\mathcal{D}_d$, $\mathcal{D}_f$, and $\mathcal{D}_i$.

Then the relevant calculation is as follows

$$\begin{aligned}
\mathsf{Adv}^{\mathsf{aeac}}_{\mathsf{SCh}}(\mathcal{D}) =& 2\Pr[\mathrm{AEAC}^{\mathcal{D}}_{\mathsf{SCh}}] - 1 = 2\Pr[\mathrm{G}_0] - 1 \\
=& 2(\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_1] + \Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_2] + \Pr[\mathrm{G}_2] - \Pr[\mathrm{G}_3] \\
& + \Pr[\mathrm{G}_3] - \Pr[\mathrm{G}_4] + \Pr[\mathrm{G}_4] - \Pr[\mathrm{G}_5] + \Pr[\mathrm{G}_5]) - 1 \\
\leq& 2(q \cdot 2^{-\mu} + \mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathcal{A}_{\mathsf{H}}) + \mathsf{Adv}^{\mathsf{ufexp}}_{\mathsf{DS}}(\mathcal{A}_{\mathsf{DS}}) + 0 + \mathsf{Adv}^{\mathsf{uniq}}_{\mathsf{DS}}(\mathcal{B}_{\mathsf{DS}})) \\
& + 2\Pr[\mathrm{INDEXP}^{\mathcal{A}_{\mathsf{PKE}}}_{\mathsf{PKE}}] - 1 \\
=& 2(q \cdot 2^{-\mu} + \mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathcal{A}_{\mathsf{H}}) + \mathsf{Adv}^{\mathsf{ufexp}}_{\mathsf{DS}}(\mathcal{A}_{\mathsf{DS}}) + \mathsf{Adv}^{\mathsf{uniq}}_{\mathsf{DS}}(\mathcal{B}_{\mathsf{DS}})) + \mathsf{Adv}^{\mathsf{indexp}}_{\mathsf{PKE}}(\mathcal{A}_{\mathsf{PKE}}).
\end{aligned}$$

To prove the first four claims we need to consider the sequence of games shown in Fig. 14. Lines of code annotated with comments are only in the specified games, all other code is common to all game. We used boxes to emphasize the lines of code annotated with comments. We use highlighting to emphasize the areas of new code.

The first three game transitions (from G_0 to G_3) will show that any ciphertext sent to u by the adversary can be rejected unless either it equals a ciphertext created by $\overline{u}$, u is already restricted, or the adversary has exposed $\overline{u}$ to steal the corresponding signing key. As part of this the transitions from G_0 to G_2 will show roughly that setting a $\mathsf{restricted}$ flag results in the two users having differing transcripts.

Claim 1, $\Pr[\mathrm{G}_0] = \Pr[\mathrm{AEAC}^{\mathcal{D}}_{\mathsf{SCh}}]$. Game G_0 was created by hardcoding the code of SCh into $\mathrm{AEAC}^{\mathcal{D}}$, flipping the position of the two if statements in Exp, and then adding some variables to help us transition to future games, which we will describe momentarily. Rather than storing the state of each user as a tuple we store the components of the state in separate variables with an underscript to denote which user's state they are from. None of these variables have any effect on the input-output behavior of G_0 so the first claim, $\Pr[\mathrm{G}_0] = \Pr[\mathrm{AEAC}^{\mathcal{D}}_{\mathsf{SCh}}]$ is immediate.

Descriptions of games G_1, G_2, and G_3. Variables t_1 through t_5 and $\vec{\Delta}_e$ are temporary variables which were added to make the code more compact. Their uses are not highlighted.

Variables r_u^{sent} and r_u^{rest} store indices corresponding to information about the ciphertexts that a user has received. In r_u^{sent} we store the last value of r_u that was included in a ciphertext sent by u (i.e. the index of the last ciphertext that u has told $\overline{u}$ they have received). This variable has no effect on the code, but will be useful for reasoning about its behavior. In r_u^{rest} we store the value held by r_u when $\mathsf{restricted}_u$ was first set or ∞ if this has not occurred.

The flag $\mathsf{unchanged}_u$ will be initialized to true and will be set false the first time that the sequence of ciphertexts received by u differs from the sequence of ciphertext sent by $\overline{u}$ while unrestricted (i.e. the first time its view is "changed" from the correct view output by $\overline{u}$). Once $\overline{u}$ is restricted **ctable** would not be set on line 39, so a the corresponding t_b in Recv would have to be false. This flag is not the exact opposite of $\mathsf{restricted}_u$ because $\mathsf{restricted}_u$ only gets set to true if the changed ciphertext was sent after an exposure. However, it is useful to note that $\mathsf{restricted}_u$ will always be false when $\mathsf{unchanged}_u$ is true.

Table $\mathbf{rctable}_u$ stores the sequence of ciphertexts that have been received and accepted by u. Table $\mathbf{sctable}_u$ stores the sequence of ciphertexts that have been sent by $\overline{u}$. This differs from the existing

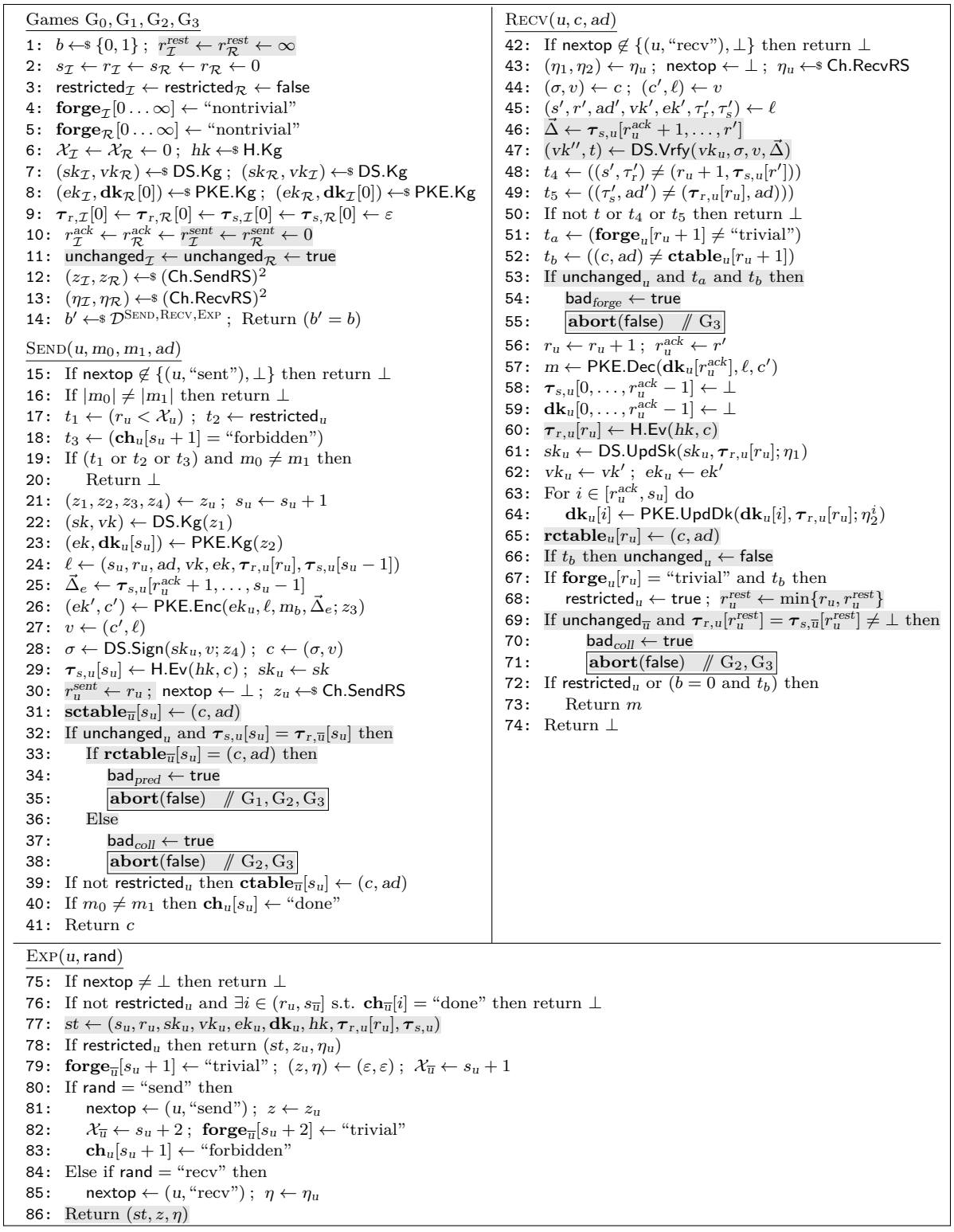

Figure 14: Games G_1, G_2, and G_3 for security proof. Lines commented with the names of games are only included in those games. Highlighting indicates new code.

table $\mathbf{ctable}_u$ because it continues to store these values even after $\mathsf{restricted}_{\overline{u}}$ has been set.

On every call to RECV, we use $\vec{\Delta}$ as a temporary variable to store the sequence of transcripts used to update the digital signature verification key.

We seek to argue that before the first time u's view is changed, the adversary should not be able to create any (c, ad) pairs that will be accepted by u unless the pair was output by $\overline{u}$ or the adversary has done an appropriate exposure of $\overline{u}$. This belief about (c, ad) pairs provided by the adversary that should be rejected is encoded in the if statement in RECV on line 53 which checks if $\mathsf{unchanged}_u$, $\mathbf{forge}_u[r_u + 1] \neq$ "trivial", and $(c, ad) \neq \mathbf{ctable}_u[r_u + 1]$. Once we reach G_3 we will simply abort if this evaluates to true.

The first two game transitions (to games G_1 and G_2) are used to rule out the possibility that the other user is restricted, but has receiving transcripts which match the sending transcripts of the current user. This possibility is captured by the if statements on lines 32 and 69. Were this possible then forgeries would be possible because after the other user is restricted the adversary can have their secrets exposed without setting the appropriate entry of $\mathbf{forge}_{\overline{u}}$ to "trivial". This will be used again at the end of the proof because if this were possible it would also lead to an attack against the security provide by the encryption scheme.

Claim 2, $\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_1] \leq q \cdot 2^{-\mu}$. To start this argument consider the flag bad_{pred}. Games G_0 and G_1 are identical until bad_{pred}. So from the fundamental lemma of game playing [8] we have $\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_1] \leq \Pr[\mathrm{G}_0 \text{ sets } \mathsf{bad}_{pred}]$. To establish the second claim we need to show that $\Pr[\mathrm{G}_0 \text{ sets } \mathsf{bad}_{pred}] \leq q \cdot 2^{-\mu}$. This will hold because an adversary can only cause bad_{pred} to be set by predicting the output of $\mathsf{Ch.Send}$ before it is generated (which requires predicting the output of $\mathsf{DS.Kg}$, $\mathsf{PKE.Kg}$, and $\mathsf{PKE.Enc}$).

This flag gets set in SEND when $\mathsf{unchanged}_u$, $\boldsymbol{\tau}_{s,u}[s_u] = \boldsymbol{\tau}_{r,\overline{u}}[s_u]$, and $\mathbf{rctable}_{\overline{u}}[s_u] = (c, ad)$ are all true. Note that because $\mathsf{unchanged}_u$ is true, it must hold that $\mathsf{restricted}_u$ is false. Thus an exposure of the randomness used by the SEND operation would set nextop to $(u, \text{"send"})$ and could thus only happen immediately before the SEND operation (and after the RECV operation during which $\mathbf{rctable}_{\overline{u}}[s_u]$ was set). This means the adversary must have predicted the output of SEND before making the call. By the properties of min-entropy, the probability this happens from any individual RECV call is at most $2^{-\mu}$ and then by a union bound we get the desired second claim.

Claim 3, $\Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_2] \leq \mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathcal{A}_{\mathsf{H}})$. For the third claim note that games G_1 and G_2 are identical until bad_{coll} so $\Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_2] \leq \Pr[\mathrm{G}_1 \text{ sets } \mathsf{bad}_{coll}]$.

When this flag is set in SEND it must hold that $\mathbf{sctable}_{\overline{u}}[s_u] \neq \mathbf{rctable}_{\overline{u}}[s_u]$ yet $\boldsymbol{\tau}_{s,u}[s_u] = \boldsymbol{\tau}_{r,\overline{u}}[s_u]$. So the corresponding ciphertexts must have formed a collision in H. (Note here that ciphertexts include the associated date, so $\mathbf{sctable}_{\overline{u}}[r^{rest}_{\overline{u}}] \neq \mathbf{rctable}_{\overline{u}}[r^{rest}_{\overline{u}}]$ implies the corresponding ciphertexts are distinct.)

When want to argue that when this flag is set in RECV it must hold that $\mathbf{rctable}_u[r^{rest}_u] \neq \mathbf{sctable}_u[r^{rest}_u]$ even though $\boldsymbol{\tau}_{r,u}[r^{rest}_u] = \boldsymbol{\tau}_{s,\overline{u}}[r^{rest}_u] \neq \perp$. Not that $\mathsf{unchanged}_{\overline{u}}$ is true, so $\mathsf{restricted}_{\overline{u}}$ must be false. If $\boldsymbol{\tau}_{r,u}[r^{rest}_u]$ was set to a non-$\perp$ value first, then bad_{coll} would have already been set true in SEND when $\boldsymbol{\tau}_{s,\overline{u}}[r^{rest}_u]$ was set. So suppose $\boldsymbol{\tau}_{s,\overline{u}}[r^{rest}_u]$ was set first and consider the first time it was set. The variable t_b was true at this time so we had $\mathbf{rctable}_u[r^{rest}_u] = (c, ad) \neq \mathbf{ctable}_u[r^{rest}_u] = \mathbf{sctable}_u[r^{rest}_u]$. (The last equality held because $\mathsf{restricted}_{\overline{u}}$ is still false.) The same reasoning as the previous paragraph implies that the corresponding ciphertexts must have formed

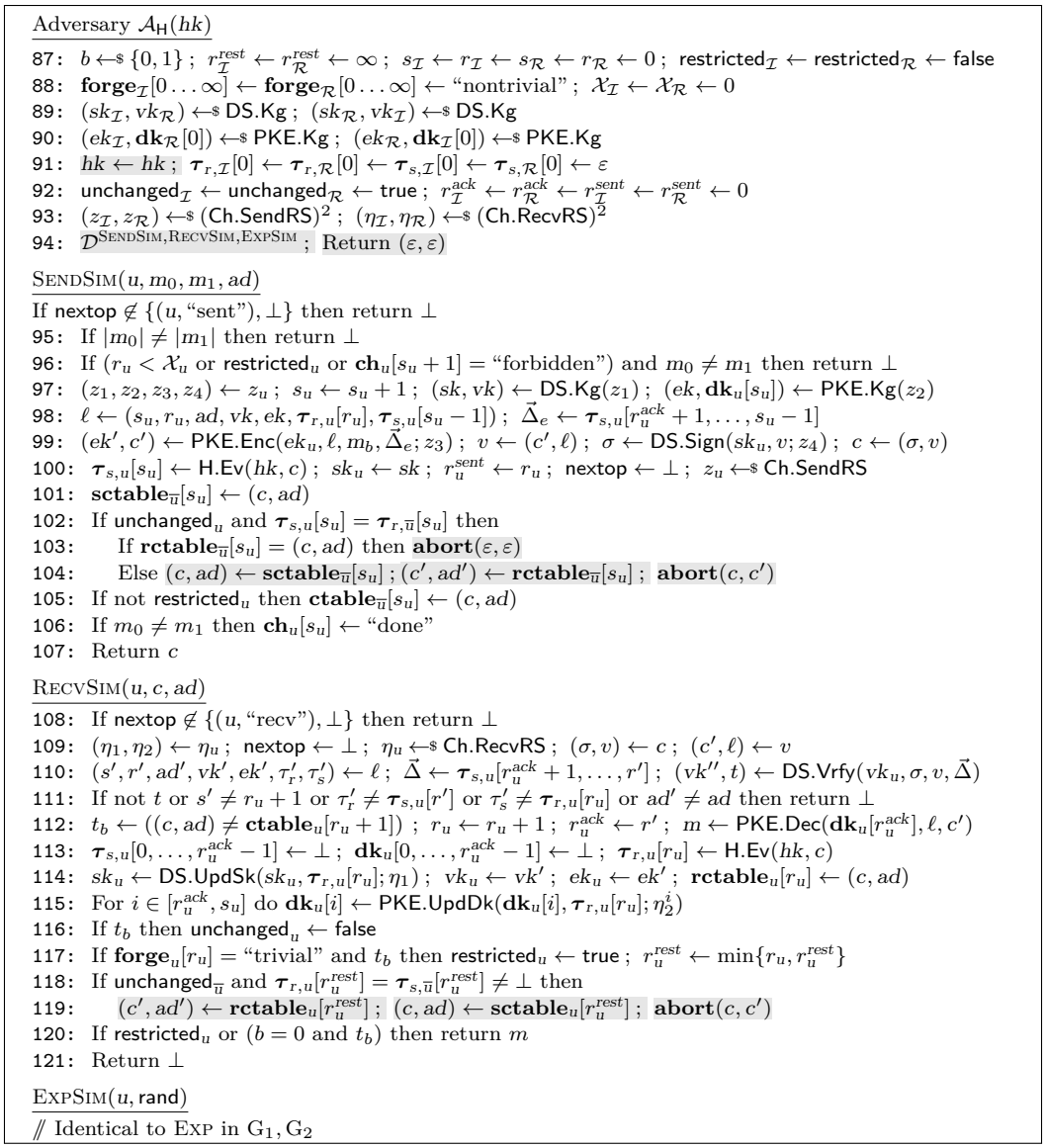

Figure 15: Adversary $\mathcal{A}_{\mathsf{H}}$ against collision resistance of H. Highlighting indicates the changes from $\mathrm{G}_1, \mathrm{G}_2$.

a collision in H.

The adversary $\mathcal{A}_{\mathsf{H}}$ shown in Fig. 15 takes advantage of this to attack the collision-resistance of H. It simulates the view of $\mathcal{D}$ and then (on line 104 or line 119) aborts and returns the collision when bad_{coll} would be set; otherwise it simply gives up and returns $(\varepsilon, \varepsilon)$. The highlighted code shows where the code of $\mathcal{A}_{\mathsf{H}}$ differs from that of G_1 and G_2. The code $hk \leftarrow hk$ is included just to make explicit that it uses the key to the hash function it was provided as input rather than sampling a new key for itself. That $\mathcal{A}_{\mathsf{H}}$ perfectly simulates the view of $\mathcal{D}$ in G_1 or G_2 until bad_{coll} is clear. From our above analysis we then have that $\mathcal{A}_{\mathsf{H}}$ succeeds in CR whenever G_1 would set bad_{coll} so $\Pr[\mathrm{G}_1 \text{ sets } \mathsf{bad}_{coll}] \leq \Pr[\mathrm{CR}^{\mathcal{A}_{\mathsf{H}}}_{\mathsf{H}}] = \mathsf{Adv}^{\mathsf{cr}}_{\mathsf{H}}(\mathcal{A}_{\mathsf{H}})$ as desired.

Adversary $\mathcal{A}_{\mathsf{DS}}^{\textsc{Upd},\textsc{Sign},\textsc{Exp}}$

$b \leftarrow\$ \{0,1\}$; $r_{\mathcal{I}}^{rest} \leftarrow r_{\mathcal{R}}^{rest} \leftarrow \infty$
$s_{\mathcal{I}} \leftarrow r_{\mathcal{I}} \leftarrow s_{\mathcal{R}} \leftarrow r_{\mathcal{R}} \leftarrow 0$
$\mathsf{restricted}_{\mathcal{I}} \leftarrow \mathsf{restricted}_{\mathcal{R}} \leftarrow \mathsf{false}$
$\mathbf{forge}_{\mathcal{I}}[0 \ldots \infty] \leftarrow$ "nontrivial"
$\mathbf{forge}_{\mathcal{R}}[0 \ldots \infty] \leftarrow$ "nontrivial"
$\mathcal{X}_{\mathcal{I}} \leftarrow \mathcal{X}_{\mathcal{R}} \leftarrow 0$; $hk \leftarrow\$ \mathsf{H.Kg}$
$vk_{\mathcal{R}} \leftarrow \textsc{NewUser}((s_{\mathcal{I}}, \mathcal{I}))$; $sk_{\mathcal{I}} \leftarrow \bot$
$vk_{\mathcal{I}} \leftarrow \textsc{NewUser}((s_{\mathcal{R}}, \mathcal{R}))$; $sk_{\mathcal{R}} \leftarrow \bot$
$(ek_{\mathcal{I}}, \mathbf{dk}_{\mathcal{R}}[0]) \leftarrow\$ \mathsf{PKE.Kg}$; $(ek_{\mathcal{R}}, \mathbf{dk}_{\mathcal{I}}[0]) \leftarrow\$ \mathsf{PKE.Kg}$
$\boldsymbol{\tau}_{r,\mathcal{I}}[0] \leftarrow \boldsymbol{\tau}_{r,\mathcal{R}}[0] \leftarrow \boldsymbol{\tau}_{s,\mathcal{I}}[0] \leftarrow \boldsymbol{\tau}_{s,\mathcal{R}}[0] \leftarrow \varepsilon$
$\mathsf{unchanged}_{\mathcal{I}} \leftarrow \mathsf{unchanged}_{\mathcal{R}} \leftarrow \mathsf{true}$
$r_{\mathcal{I}}^{ack} \leftarrow r_{\mathcal{R}}^{ack} \leftarrow r_{\mathcal{I}}^{sent} \leftarrow r_{\mathcal{R}}^{sent} \leftarrow 0$
$(z_{\mathcal{I}}, z_{\mathcal{R}}) \leftarrow\$ (\mathsf{Ch.SendRS})^2$
$(\eta_{\mathcal{I}}, \eta_{\mathcal{R}}) \leftarrow\$ (\mathsf{Ch.RecvRS})^2$
$\mathcal{D}^{\textsc{SendSim},\textsc{RecvSim},\textsc{ExpSim}}$
Return $(\varepsilon, \varepsilon, \varepsilon, (\varepsilon))$

$\textsc{SendSim}(u, m_0, m_1, ad)$

If $\mathsf{nextop} \notin \{(u, \text{"sent"}), \bot\}$ then return $\bot$
If $|m_0| \neq |m_1|$ then return $\bot$
$t_1 \leftarrow (r_u < \mathcal{X}_u)$; $t_2 \leftarrow \mathsf{restricted}_u$
$t_3 \leftarrow (\mathbf{ch}_u[s_u + 1] = \text{"forbidden"})$
If (t_1 or t_2 or t_3) and $m_0 \neq m_1$ then return $\bot$
$(z_1, z_2, z_3, z_4) \leftarrow z_u$; $s_u \leftarrow s_u + 1$
If $\mathsf{nextop} \neq (u, \text{"sent"})$ then
  $vk \leftarrow \textsc{NewUser}((s_u, u))$; $sk \leftarrow \bot$
Else $(sk, vk) \leftarrow \mathsf{DS.Kg}(z_1)$
$(ek, \mathbf{dk}_u[s_u]) \leftarrow \mathsf{PKE.Kg}(z_2)$
$\ell \leftarrow (s_u, r_u, ad, vk, ek, \boldsymbol{\tau}_{r,u}[r_u], \boldsymbol{\tau}_{s,u}[s_u - 1])$
$\vec{\Delta}_e \leftarrow \boldsymbol{\tau}_{s,u}[r_u^{ack} + 1, \ldots, s_u - 1]$
$(ek', c') \leftarrow \mathsf{PKE.Enc}(ek_u, \ell, m_b, \vec{\Delta}_e; z_3)$
$v \leftarrow (c', \ell)$
If $sk_u = \bot$ then $\sigma \leftarrow \textsc{Sign}((s_u - 1, u), v)$
Else $\sigma \leftarrow \mathsf{DS.Sign}(sk_u, v; z_4)$
$c \leftarrow (\sigma, v)$; $\boldsymbol{\tau}_{s,u}[s_u] \leftarrow \mathsf{H.Ev}(hk, c)$; $sk_u \leftarrow sk$
$r_u^{sent} \leftarrow r_u$; $\mathsf{nextop} \leftarrow \bot$; $z_u \leftarrow\$ \mathsf{Ch.SendRS}$
$\mathbf{sctable}_{\overline{u}}[s_u] \leftarrow (c, ad)$
If $\mathsf{unchanged}_u$ and $\boldsymbol{\tau}_{s,u}[s_u] = \boldsymbol{\tau}_{r,\overline{u}}[s_u]$ then
  $\mathbf{abort}(\varepsilon, \varepsilon, \varepsilon, (\varepsilon))$
If not $\mathsf{restricted}_u$ then $\mathbf{ctable}_{\overline{u}}[s_u] \leftarrow (c, ad)$
If $m_0 \neq m_1$ then $\mathbf{ch}_u[s_u] \leftarrow$ "done"
Return c

$\textsc{RecvSim}(u, c, ad)$

If $\mathsf{nextop} \notin \{(u, \text{"recv"}), \bot\}$ then return $\bot$
$(\eta_1, \eta_2) \leftarrow \eta_u$; $\mathsf{nextop} \leftarrow \bot$
$\eta_u \leftarrow\$ \mathsf{Ch.RecvRS}$
$(\sigma, v) \leftarrow c$; $(c', \ell) \leftarrow v$
$(s', r', ad', vk', ek', \tau_r', \tau_s') \leftarrow \ell$
$\vec{\Delta} \leftarrow \boldsymbol{\tau}_{s,u}[r_u^{ack} + 1, \ldots, r']$
$(vk'', t) \leftarrow \mathsf{DS.Vrfy}(vk_u, \sigma, v, \vec{\Delta})$
$t_4 \leftarrow ((s', \tau_r') \neq (r_u + 1, \boldsymbol{\tau}_{s,u}[r']))$
$t_5 \leftarrow ((\tau_s', ad') \neq (\boldsymbol{\tau}_{r,u}[r_u], ad)))$
If not t or t_4 or t_5 then return $\bot$
$t_a \leftarrow (\mathbf{forge}_u[r_u + 1] \neq$ "trivial")
$t_b \leftarrow ((c, ad) \neq \mathbf{ctable}_u[r_u + 1])$
If $\mathsf{unchanged}_u$ and t_a and t_b then
  $\mathbf{abort}((r_u, \overline{u}), \sigma, v, \vec{\Delta})$
$r_u \leftarrow r_u + 1$; $r_u^{ack} \leftarrow r'$
$m \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}_u[r_u^{ack}], \ell, c')$
$\boldsymbol{\tau}_{s,u}[0, \ldots, r_u^{ack} - 1] \leftarrow \bot$
$\mathbf{dk}_u[0, \ldots, r_u^{ack} - 1] \leftarrow \bot$
$\boldsymbol{\tau}_{r,u}[r_u] \leftarrow \mathsf{H.Ev}(hk, c)$
If $sk_u = \bot$ then $\textsc{Upd}((s_u, u), \boldsymbol{\tau}_{r,u}[r_u])$
Else $sk_u \leftarrow \mathsf{DS.UpdSk}(sk_u, \boldsymbol{\tau}_{r,u}[r_u]; \eta_1)$
$vk_u \leftarrow vk'$; $ek_u \leftarrow ek'$
For $i \in [r_u^{ack}, s_u]$ do
  $\mathbf{dk}_u[i] \leftarrow \mathsf{PKE.UpdDk}(\mathbf{dk}_u[i], \boldsymbol{\tau}_{r,u}[r_u]; \eta_1^i)$
$\mathbf{rctable}_u[r_u] \leftarrow (c, ad)$
If t_b then $\mathsf{unchanged}_u \leftarrow \mathsf{false}$
If $\mathbf{forge}_u[r_u] =$ "trivial" and t_b then
  $\mathsf{restricted}_u \leftarrow \mathsf{true}$; $r_u^{rest} \leftarrow \min\{r_u, r_u^{rest}\}$
If $\mathsf{unchanged}_{\overline{u}}$ and $\boldsymbol{\tau}_{r,u}[r_u^{rest}] = \boldsymbol{\tau}_{s,\overline{u}}[r_u^{rest}] \neq \bot$ then
  $\mathbf{abort}(\varepsilon, \varepsilon, \varepsilon, (\varepsilon))$
If $\mathsf{restricted}_u$ or ($b = 0$ and t_b) then
  Return m
Return $\bot$

$\textsc{ExpSim}(u, \mathsf{rand})$

If $\mathsf{nextop} \neq \bot$ then return $\bot$
If not $\mathsf{restricted}_u$ and $\exists i \in (r_u, s_{\overline{u}}]$ s.t. $\mathbf{ch}_{\overline{u}}[i] =$ "done" then return $\bot$
If $sk_u = \bot$ then $sk_u \leftarrow \textsc{Exp}((s_u, u))$
$st \leftarrow (s_u, r_u, sk_u, vk_u, ek_u, \mathbf{dk}_u, hk, \boldsymbol{\tau}_{r,u}[r_u], \boldsymbol{\tau}_{s,u})$
If $\mathsf{restricted}_u$ then return (st, z_u, η_u)
$\mathbf{forge}_{\overline{u}}[s_u + 1] \leftarrow$ "trivial" ; $(z, \eta) \leftarrow (\varepsilon, \varepsilon)$; $\mathcal{X}_{\overline{u}} \leftarrow s_u + 1$
If $\mathsf{rand} =$ "send" then
  $\mathsf{nextop} \leftarrow (u, \text{"send"})$; $z \leftarrow z_u$; $\mathcal{X}_{\overline{u}} \leftarrow s_u + 2$; $\mathbf{forge}_{\overline{u}}[s_u + 2] \leftarrow$ "trivial"
  $\mathbf{ch}_u[s_u + 1] \leftarrow$ "forbidden"
Else if $\mathsf{rand} =$ "recv" then
  $\mathsf{nextop} \leftarrow (u, \text{"recv"})$; $\eta \leftarrow \eta_u$
Return (st, z, η)

Figure 16: Adversary $\mathcal{A}_{\mathsf{DS}}$ attacking DS. Highlighting indicates changes from $\mathrm{G}_2, \mathrm{G}_3$

Claim 4, $\Pr[\mathrm{G}_2] - \Pr[\mathrm{G}_3] \leq \mathsf{Adv}_{\mathsf{DS}}^{\mathsf{ufexp}}(\mathcal{A}_{\mathsf{DS}})$. Finally we can give our reduction to the security of DS to complete this stage of the proof. Game G_2 and G_3 are identical until bad_{forge}, so $\Pr[\mathrm{G}_2] - \Pr[\mathrm{G}_3] \leq$

$\Pr[\mathrm{G}_2 \text{ sets } \mathsf{bad}_{forge}]$. The adversary $\mathcal{A}_{\mathsf{DS}}$ shown in Fig. 16 against the security of DS uses its oracles to simulate the view of $\mathcal{D}$ and then (on line 175) aborts and returns the forgery if bad_{forge} would ever be set. The highlighting shows where the code of $\mathcal{A}_{\mathsf{DS}}$ differs from that of games G_2 and G_3. Below we verify that $\mathcal{A}_{\mathsf{DS}}$ correctly simulates the view of $\mathcal{D}$ in these games (until the time that it aborts).

Now consider a call to RecvSim$(u, \cdot, \cdot)$ by $\mathcal{D}$ that results in $\mathcal{A}_{\mathsf{DS}}$ aborting. Note that $\mathcal{A}_{\mathsf{DS}}$ uses user identifiers of the form $\Lambda = (s, v)$ for the key created by user v with the s-th ciphertext it sends. We seek to argue that $\mathcal{A}_{\mathsf{DS}}$ wins UFEXP whenever it aborts, so we need to show that (1) $vk[(r_u, \overline{u})] \neq \bot$, (2) $\mathsf{win} = \mathsf{true}$, and (3) $\mathsf{cheated} = \mathsf{false}$. Then because $\mathcal{A}_{\mathsf{DS}}$ aborts whenever bad_{forge} would be set we get $\Pr[\mathrm{G}_2 \text{ sets } \mathsf{bad}_{forge}] \leq \Pr[\mathrm{UFEXP}^{\mathcal{A}_{\mathsf{DS}}}_{\mathsf{DS}}] = \mathsf{Adv}^{\mathsf{ufexp}}_{\mathsf{DS}}(\mathcal{A}_{\mathsf{DS}})$ as desired.

Successful aborts by $\mathcal{A}_{\mathsf{DS}}$. First note that $\mathsf{unchanged}_u$ is true because of line 174, so vk_u must equal the verification key that was part of the ciphertext sent by $\overline{u}$ when $s_{\overline{u}}$ was equal to the current value r_u. This was output by NewUser$((s_{\overline{u}}, \overline{u}))$ on line 145 unless it held that $\mathsf{nextop} = (\overline{u}, \text{“sent”})$. The only way the latter can hold is if nextop was just set in ExpSim, but then $\mathbf{forge}_{\overline{u}}[s_{\overline{u}} + 1] = \mathbf{forge}_{\overline{u}}[r_u + 1]$ would have been set to “trivial” contradicting that t_a holds in RecvSim. (Here note that $s_{\overline{u}}$ was incremented at the start of the SendSim call, so $s_{\overline{u}} + 2$ during the exposure was equal to the value of $s_{\overline{u}} + 1$ while sending.) So vk_u was output by NewUser$((s_{\overline{u}}, \overline{u}))$ and $vk[(r_u, \overline{u})] \neq \bot$ in UFEXP. This is (1).

That $\mathsf{win} = \mathsf{true}$ in UFEXP follows immediately from the above and the fact that t was true in RecvSim (see line 171). This is (2).

We complete this argument by showing (3) that if $\mathsf{cheated} = \mathsf{true}$ holds, then it must hold that $\mathsf{restricted}_{\overline{u}}$ is true and $\boldsymbol{\tau}_{r,\overline{u}}[r^{rest}_{\overline{u}}] = \boldsymbol{\tau}_{s,u}[r^{rest}_{\overline{u}}]$ which cannot be the case because of lines 158 and 191 and the fact that $\mathsf{unchanged}_u$ is true. If $\mathsf{cheated}$ is true then either $(\sigma, v, \vec{\Delta}) = (\sigma^*[(r_u, \overline{u})], m^*[(r_u, \overline{u})], \vec{\Delta}^*[(r_u, \overline{u})])$ or $\vec{\Delta}'[(r_u, \overline{u})] \sqsubseteq \vec{\Delta}$ holds.

We start with the former case. The variables $\sigma^*[(r_u, \overline{u})]$, $m^*[(r_u, \overline{u})]$, and $\vec{\Delta}^*[(r_u, \overline{u})]$ would have been defined by the Sign call on line 152 when $\overline{u}$ was sending a ciphertext and $s_{\overline{u}} = r_u + 1$. Variables $\sigma^*[(r_u, \overline{u})]$ and $m^*[(r_u, \overline{u})]$ thus uniquely define the ciphertext and associated data stored in $\mathbf{sctable}_u[s_{\overline{u}}]$ on line 156 (note that the ad component is uniquely defined because it is included as part of the ciphertext). So these variables equaling the σ and v returned by $\mathcal{A}_{\mathsf{DS}}$ when it aborts means that $\mathbf{sctable}_u[r_u + 1] = (c, ad) \neq \mathbf{ctable}_u[r_u + 1]$ where the inequality is from t_b holding in RecvSim. This implies that $\mathsf{restricted}_{\overline{u}}$ was true when $\mathbf{sctable}_u[r_u + 1]$ and $\mathbf{ctable}_u[r_u + 1]$ were set on lines 156 and 159 (so, in particular, $\mathbf{ctable}_u[r_u + 1]$ was not set). We will show the following sequence of equalities holds $\boldsymbol{\tau}_{r,\overline{u}}[r^{rest}_{\overline{u}}] = \vec{\Delta}^*[(r_u, \overline{u})][r^{rest}_{\overline{u}} - r^{sent}_{\overline{u}}] = \vec{\Delta}[r^{rest}_{\overline{u}} - r^{sent}_{\overline{u}}] = \boldsymbol{\tau}_{s,u}[r^{rest}_{\overline{u}}]$, where $\vec{\Delta}^*[(r_u, \overline{u})]$ and $\vec{\Delta}$ are indexed starting from 1. The latter two equalities are immediate from our assumption that $\vec{\Delta}^*[(r_u, \overline{u})] = \vec{\Delta}$ and the way $\vec{\Delta}$ is defined on line 167. Because $\overline{u}$ was restricted during the SendSim query that defined $\vec{\Delta}^*[(r_u, \overline{u})]$ it must have held that $r^{rest}_{\overline{u}}$ was less than or equal to the value of $r_{\overline{u}}$. It must also have held that $r^{rest}_{\overline{u}}$ was strictly greater than $r^{sent}_{\overline{u}}$ because $\overline{u}$ was not restricted during their prior send operation (because u received this prior ciphertext and $\mathsf{unchanged}_u$ still holds). So during the receive operation that set $\mathsf{restricted}_{\overline{u}}$ and all other receive operations of $\overline{u}$ after they sent that prior ciphertext it could not have held that $sk_{\overline{u}} \neq \bot$ because that necessitates $\mathbf{forge}_u[r_u + 1] =$ “trivial”, contradicting our assumption that $\mathcal{A}_{\mathsf{DS}}$ aborted. So line 181 will have resulted in $\vec{\Delta}_s[(r_u, \overline{u})][1, \ldots, r^{rest}_{\overline{u}} - r^{sent}_{\overline{u}}] = \boldsymbol{\tau}_{r,\overline{u}}[r^{sent}_{\overline{u}} + 1, \ldots, r^{rest}_{\overline{u}}]$. Then the first equality follows because $\vec{\Delta}^*[(r_u, \overline{u})]$ is later set to equal $\vec{\Delta}_s[(r_u, \overline{u})]$ during the sending operation.

Consider the latter case that $\vec{\Delta}'[(r_u, \overline{u})] \sqsubseteq \vec{\Delta}$. Then $\vec{\Delta}'[(r_u, \overline{u})] \neq \perp$ so the adversary must have exposed $\overline{u}$ when $s_{\overline{u}}$ was equal to the current value of r_u. By assumption $\mathbf{forge}_u[r_u + 1] \neq$ "trivial" so this exposure must have been done when $\mathsf{restricted}_{\overline{u}}$ was true. The same arguments above apply to give that $\vec{\Delta}_s[(r_u, \overline{u})][1, \ldots, r^{rest}_{\overline{u}} - r^{sent}_{\overline{u}}] = \boldsymbol{\tau}_{r,\overline{u}}[r^{sent}_{\overline{u}} + 1, \ldots, r^{rest}_{\overline{u}}]$. Then later during the exposure $\vec{\Delta}'[(r_u, \overline{u})]$ is set to equal $\vec{\Delta}_s[(r_u, \overline{u})]$. This results in the sequence of equalities $\boldsymbol{\tau}_{r,\overline{u}}[r^{rest}_{\overline{u}}] = \vec{\Delta}'[(r_u, \overline{u})][r^{rest}_{\overline{u}} - r^{sent}_{\overline{u}}] = \vec{\Delta}[r^{rest}_{\overline{u}} - r^{sent}_{\overline{u}}] = \boldsymbol{\tau}_{s,u}[r^{rest}_{\overline{u}}]$ from the assumption that $\vec{\Delta}'[(r_u, \overline{u})] \sqsubseteq \vec{\Delta}$ and how $\vec{\Delta}$ is defined.

Now we show $\boldsymbol{\tau}_{r,\overline{u}}[r^{rest}_{\overline{u}}] = \boldsymbol{\tau}_{s,u}[r^{rest}_{\overline{u}}]$. If $\vec{\Delta}'_{\overline{u}}[r_u] \sqsubseteq \vec{\Delta}$ holds, $\overline{u}$ must have been restricted before $\vec{\Delta}'_{\overline{u}}[r_u]$ was set (because t_a is true, so line 79 must not have been executed). Thus the upper bound index used to set $\vec{\Delta}'_{\overline{u}}[r_u]$ was at least $r^{rest}_{\overline{u}}$. The lower bound index was equal to $r^{ack}_u + 1$; from the argument about, $r^{rest}_{\overline{u}}$ is at least this large. So it must hold that $\boldsymbol{\tau}_{r,\overline{u}}[r^{rest}_{\overline{u}}] = \boldsymbol{\tau}_{s,u}[r^{rest}_{\overline{u}}]$.

Correct simulation by $\mathcal{A}_{\mathsf{DS}}$. The correctness of $\mathcal{A}_{\mathsf{DS}}$ follows from the fact that none of $\mathcal{A}_{\mathsf{DS}}$'s oracles will abort early and return $\perp$. Recall again that $\mathcal{A}_{\mathsf{DS}}$ uses user identifiers of the form $\Lambda = (s_u, u)$.

Queries to NewUser are made in SendSim only after s_u has been incremented, so such queries will never be made for which $sk[\Lambda] \neq \perp$. For other oracles, note that $sk[(i, v)] = \perp$ will only hold if $i > s_v$ or if $\mathsf{nextop} = (v, \text{"send"})$ when the i-th ciphertext was sent by user v. The former case never occurs because oracle queries are only made with $i = s_u$ or $i = s_u - 1$. The latter case corresponds to $\mathcal{D}$ having exposed the randomness underlying this sending call, so $\mathcal{A}_{\mathsf{DS}}$ simply samples the signature keys for itself. Then $sk_u \neq \perp$ so $\mathcal{A}_{\mathsf{DS}}$ will not make any further oracle queries with $\Lambda = (i, v)$.

The last possibility is $\mathcal{A}_{\mathsf{DS}}$ making a query $\textsc{Sign}(\Lambda, m)$ when $\vec{\Delta}^*[\Lambda] \neq \perp$. As with NewUser, queries to Sign are made in SendSim only after s_u has been incremented, so $\mathcal{A}_{\mathsf{DS}}$ will never make two signing queries with the same Λ implying that this possibility will not occur.

Claim 5, $\Pr[\mathrm{G}_3] = \Pr[\mathrm{G}_4]$. To start the second stage of the proof consider games G_4 and G_5 shown in Fig. 17. We claim that the input-output behavior of G_4 is identical to G_3. We made three types of changes to G_3 to create G_4. First we generally cleaned things up by removing variables $\vec{\Delta}_u$, $\mathbf{rctable}_u$, $\mathbf{sctable}_u$, and r^{sent}_u which had no effect on the code and by simplifying the code to just abort when the if statements on lines 32, 53, or 69 evaluate to true, giving lines 224, 233, and 243. Next, we added the lines 234 through 237 for the transition to game G_5. Finally, we both replaced $\mathsf{unchanged}_u$ with (not $\mathsf{restricted}_u$) and removed the unnecessary parts of the if statements on lines 67 and 72 to obtain lines 242 and 244. These last changes require some justification.

First we claim that in G_3 when $\mathsf{unchanged}_u$ gets set false during a call to Recv, $\mathsf{restricted}_u$ will get set true during the same call. The converse is clear, so this establishes that they always have opposite truth values. The first time $\mathsf{unchanged}_u$ is set it must hold that $(c, ad) \neq \mathbf{ctable}_u[r_u]$ and that $\mathsf{unchanged}_u$ was true at the beginning of the execution of Recv. Then because the if statement on line 53 must have evaluated to false we have that $\mathbf{forge}_u[r_u] =$ "trivial" on line 67 so $\mathsf{restricted}_u$ will also be set.

For simplifying lines 67 and 72 note that if $(c, ad) \neq \mathbf{ctable}_u[r_u]$ then either $\mathsf{restricted}_u = \mathsf{true}$ already held at the beginning of this execution of Recv or the if statement on line 53 implies that $\mathbf{forge}_u[r_u + 1] =$ "trivial" (before r_u has been incremented) so $\mathsf{restricted}_u$ will be set. This allows to simplify both if statements. Thus the fifth claim, $\Pr[\mathrm{G}_3] = \Pr[\mathrm{G}_4]$, holds.

Games $\mathrm{G}_4, \mathrm{G}_5$

$b \leftarrow\!\!\$ \{0,1\}$; $r^{rest}_{\mathcal{I}} \leftarrow r^{rest}_{\mathcal{R}} \leftarrow \infty$; $s_{\mathcal{I}} \leftarrow r_{\mathcal{I}} \leftarrow s_{\mathcal{R}} \leftarrow r_{\mathcal{R}} \leftarrow 0$; $\mathsf{restricted}_{\mathcal{I}} \leftarrow \mathsf{restricted}_{\mathcal{R}} \leftarrow \mathsf{false}$
$\mathbf{forge}_{\mathcal{I}}[0 \dots \infty] \leftarrow \mathbf{forge}_{\mathcal{R}}[0 \dots \infty] \leftarrow$ "nontrivial" ; $\mathcal{X}_{\mathcal{I}} \leftarrow \mathcal{X}_{\mathcal{R}} \leftarrow 0$
$(sk_{\mathcal{I}}, vk_{\mathcal{R}}) \leftarrow\!\!\$ \mathsf{DS.Kg}$; $(sk_{\mathcal{R}}, vk_{\mathcal{I}}) \leftarrow\!\!\$ \mathsf{DS.Kg}$
$(ek_{\mathcal{I}}, \mathbf{dk}_{\mathcal{R}}[0]) \leftarrow\!\!\$ \mathsf{PKE.Kg}$; $(ek_{\mathcal{R}}, \mathbf{dk}_{\mathcal{I}}[0]) \leftarrow\!\!\$ \mathsf{PKE.Kg}$
$hk \leftarrow\!\!\$ \mathsf{H.Kg}$; $\boldsymbol{\tau}_{r,\mathcal{I}}[0] \leftarrow \boldsymbol{\tau}_{r,\mathcal{R}}[0] \leftarrow \boldsymbol{\tau}_{s,\mathcal{I}}[0] \leftarrow \boldsymbol{\tau}_{s,\mathcal{R}}[0] \leftarrow \varepsilon$; $r^{ack}_{\mathcal{I}} \leftarrow r^{ack}_{\mathcal{R}} \leftarrow 0$
$(z_{\mathcal{I}}, z_{\mathcal{R}}) \leftarrow\!\!\$ (\mathsf{Ch.SendRS})^2$; $(\eta_{\mathcal{I}}, \eta_{\mathcal{R}}) \leftarrow\!\!\$ (\mathsf{Ch.RecvRS})^2$
$b' \leftarrow\!\!\$ \mathcal{D}^{\textsc{Send},\textsc{Recv},\textsc{Exp}}$; Return $(b' = b)$

$\textsc{Send}(u, m_0, m_1, ad)$

If $\mathsf{nextop} \neq (u, \text{"send"})$ and $\mathsf{nextop} \neq \perp$ then return $\perp$
If $|m_0| \neq |m_1|$ then return $\perp$
If ($r_u < \mathcal{X}_u$ or $\mathsf{restricted}_u$ or $\mathbf{ch}_u[s_u + 1] =$ "forbidden") and $m_0 \neq m_1$ then
    Return $\perp$
$(z_1, z_2, z_3, z_4) \leftarrow z_u$
$s_u \leftarrow s_u + 1$; $(sk, vk) \leftarrow \mathsf{DS.Kg}(z_1)$; $(ek, \mathbf{dk}_u[s_u]) \leftarrow \mathsf{PKE.Kg}(z_2)$
$\ell \leftarrow (s_u, r_u, ad, vk, ek, \boldsymbol{\tau}_{r,u}[r_u], \boldsymbol{\tau}_{s,u}[s_u - 1])$
$(ek', c') \leftarrow \mathsf{PKE.Enc}(ek_u, \ell, m_b, \boldsymbol{\tau}_{s,u}[r^{ack}_u + 1, \dots, s_u - 1]; z_3)$
$v \leftarrow (c', \ell)$; $\sigma \leftarrow \mathsf{DS.Sign}(sk_u, v; z_4)$; $c \leftarrow (\sigma, v)$; $\boldsymbol{\tau}_{s,u}[s_u] \leftarrow \mathsf{H.Ev}(hk, c)$; $sk_u \leftarrow sk$
$\mathsf{nextop} \leftarrow \perp$; $z_u \leftarrow\!\!\$ \mathsf{Ch.SendRS}$
If not $\mathsf{restricted}_u$ and $\boldsymbol{\tau}_{s,u}[s_u] = \boldsymbol{\tau}_{r,\overline{u}}[s_u]$ then $\mathbf{abort}(\mathsf{false})$
If not $\mathsf{restricted}_u$ then $\mathbf{ctable}_{\overline{u}}[s_u] \leftarrow (c, ad)$
If $m_0 \neq m_1$ then $\mathbf{ch}_u[s_u] \leftarrow$ "done"
Return c

$\textsc{Recv}(u, c, ad)$

If $\mathsf{nextop} \neq (u, \text{"recv"})$ and $\mathsf{nextop} \neq \perp$ then return $\perp$
$(\eta_1, \eta_2) \leftarrow \eta_u$; $\mathsf{nextop} \leftarrow \perp$; $\eta_u \leftarrow\!\!\$ \mathsf{Ch.RecvRS}$
$(\sigma, v) \leftarrow c$; $(c', \ell) \leftarrow v$; $(s', r', ad', vk', ek', \tau'_r, \tau'_s) \leftarrow \ell$
$(vk'', t) \leftarrow \mathsf{DS.Vrfy}(vk_u, \sigma, v, \boldsymbol{\tau}_s[r^{ack}_u + 1, \dots, r'])$
If not t or $s' \neq r_u + 1$ or $\tau'_r \neq \boldsymbol{\tau}_{s,u}[r']$ or $\tau'_s \neq \boldsymbol{\tau}_{r,u}[r_u]$ or $ad' \neq ad$ then return $\perp$
If not $\mathsf{restricted}_u$ and $\mathbf{forge}_u[r_u + 1] \neq$ "trivial" and $(c, ad) \neq \mathbf{ctable}_u[r_u + 1]$ then $\mathbf{abort}(\mathsf{false})$
$((\sigma', v'), ad) \leftarrow \mathbf{ctable}_u[r_u + 1]$
If not $\mathsf{restricted}_u$ and $v = v'$ and $\sigma \neq \sigma'$ then
    $\mathsf{bad}_{uniq} \leftarrow \mathsf{true}$
    $\boxed{\mathbf{abort}(\mathsf{false}) \quad // \ \mathrm{G}_5}$
$r_u \leftarrow r_u + 1$; $r^{ack}_u \leftarrow r'$; $m \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}_u[r^{ack}_u], \ell, c')$
$\boldsymbol{\tau}_{s,u}[0, \dots, r^{ack}_u - 1] \leftarrow \perp$; $\mathbf{dk}_u[0, \dots, r^{ack}_u - 1] \leftarrow \perp$; $\boldsymbol{\tau}_{r,u}[r_u] \leftarrow \mathsf{H.Ev}(hk, c)$
$sk_u \leftarrow \mathsf{DS.UpdSk}(sk_u, \boldsymbol{\tau}_{r,u}[r_u]; \eta_1)$; $vk_u \leftarrow vk'$; $ek_u \leftarrow ek'$
For $i \in [r^{ack}_u, s_u]$ do $\mathbf{dk}_u[i] \leftarrow \mathsf{PKE.UpdDk}(\mathbf{dk}_u[i], \boldsymbol{\tau}_{r,u}[r_u]; \eta^i_2)$
If $(c, ad) \neq \mathbf{ctable}_u[r_u]$ then $\mathsf{restricted}_u \leftarrow \mathsf{true}$; $r^{rest}_u \leftarrow \min\{r_u, r^{rest}_u\}$
If not $\mathsf{restricted}_{\overline{u}}$ and $\boldsymbol{\tau}_{r,u}[r^{rest}_u] = \boldsymbol{\tau}_{s,\overline{u}}[r^{rest}_u] \neq \perp$ then $\mathbf{abort}(\mathsf{false})$
If $\mathsf{restricted}_u$ then return m
Return $\perp$

$\textsc{Exp}(u, \mathsf{rand})$

If $\mathsf{nextop} \neq \perp$ then return $\perp$
If not $\mathsf{restricted}_u$ and $\exists i \in (r_u, s_{\overline{u}}]$ s.t. $\mathbf{ch}_{\overline{u}}[i] =$ "done" then
    Return $\perp$
$st \leftarrow (s_u, r_u, sk_u, vk_u, ek_u, \mathbf{dk}_u, hk, \boldsymbol{\tau}_{r,u}[r_u], \boldsymbol{\tau}_{s,u})$
If $\mathsf{restricted}_u$ then return (st, z_u, η_u)
$\mathbf{forge}_{\overline{u}}[s_u + 1] \leftarrow$ "trivial" ; $(z, \eta) \leftarrow (\varepsilon, \varepsilon)$; $\mathcal{X}_{\overline{u}} \leftarrow s_u + 1$
If $\mathsf{rand} =$ "send" then
    $\mathsf{nextop} \leftarrow (u, \text{"send"})$; $z \leftarrow z_u$
    $\mathcal{X}_{\overline{u}} \leftarrow s_u + 2$; $\mathbf{forge}_{\overline{u}}[s_u + 2] \leftarrow$ "trivial"
    $\mathbf{ch}_u[s_u + 1] \leftarrow$ "forbidden"
Else if $\mathsf{rand} =$ "recv" then
    $\mathsf{nextop} \leftarrow (u, \text{"recv"})$; $\eta \leftarrow \eta_u$
Return (st, z, η)

Figure 17: Games G_4 and G_5 for security proof. Lines commented with the names of games are only included in those games. Highighted codes indicates changes from G_3.

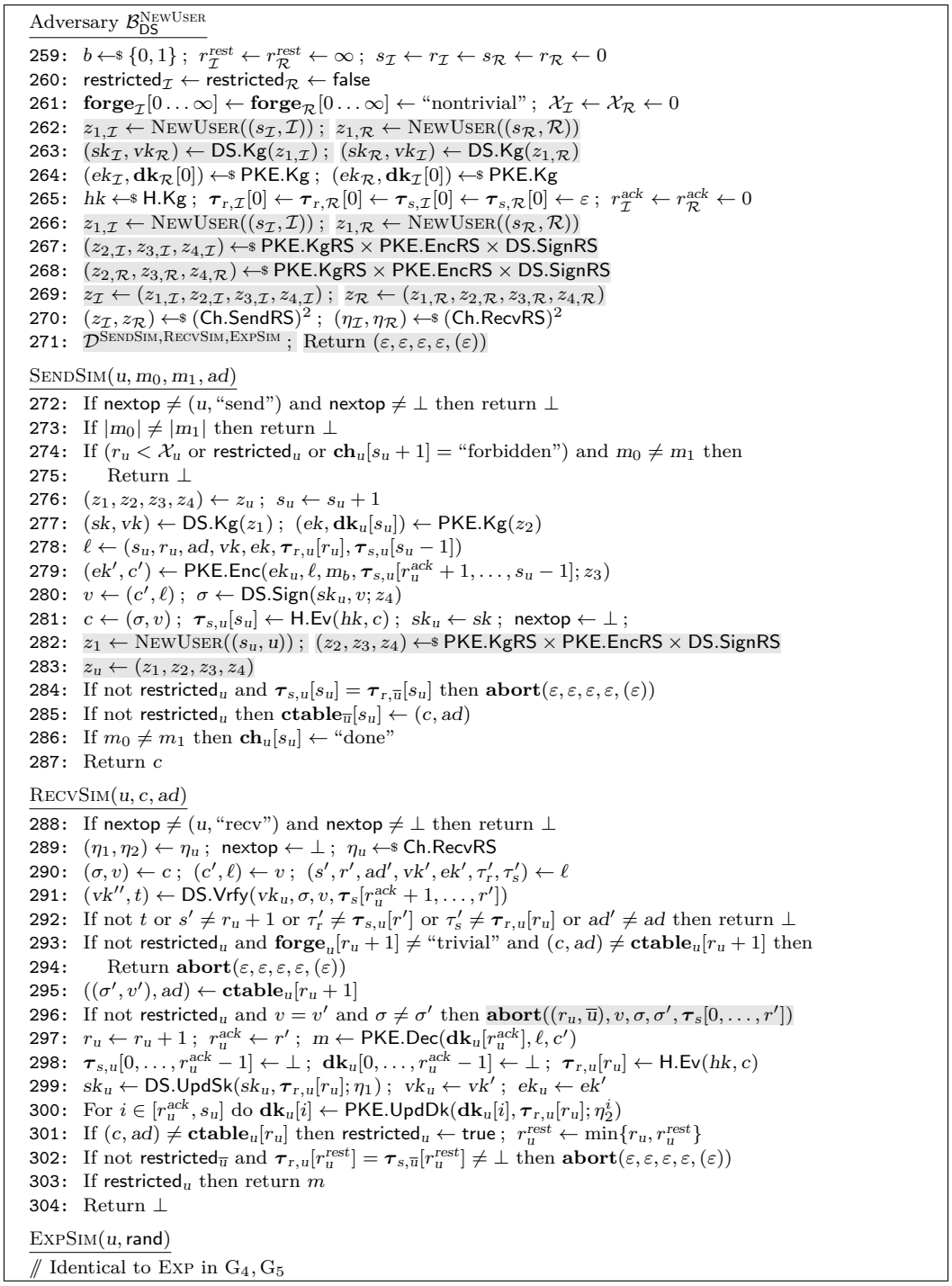

Figure 18: Adversary $\mathcal{B}_{\mathsf{DS}}$ attacking DS. Highlighting indicates changes from G_4, G_5.

Claim 6, $\Pr[G_4] - \Pr[G_5] \leq \mathsf{Adv}_{\mathsf{DS}}^{\mathsf{uniq}}(\mathcal{B}_{\mathsf{DS}})$. Games G_4 and G_5 are identical until bad_{uniq} so the fundamental lemma of game playing gives $\Pr[G_4] - \Pr[G_5] \leq \Pr[G_4 \text{ sets } \mathsf{bad}_{uniq}]$. Because bad_{uniq} only gets set if $\mathcal{D}$ has found a signature $\sigma' \neq \sigma$ which verifies correctly for v this can be transformed immediately to an attack on the UNIQ security of DS. This is shown by the adversary $\mathcal{B}_{\mathsf{DS}}$ in Fig. 18. It simulates the view of $\mathcal{D}$ by using NewUser to create the randomness used for digital

signature key generation and computing everything else for itself. On line 296 it aborts and returns the signature collision found by $\mathcal{D}$ whenever bad_{uniq} would be set in G_4. It is clear that $\mathcal{B}_{\mathsf{DS}}$ correctly simulates the view of $\mathcal{D}$ and we will momentarily justify that it wins whenever it aborts, so $\Pr[G_4 \text{ sets } \mathsf{bad}_{uniq}] = \Pr[\text{UNIQ}^{\mathcal{B}_{\mathsf{DS}}}_{\mathsf{DS}}] = \mathsf{Adv}^{\mathsf{uniq}}_{\mathcal{B}_{\mathsf{DS}}}(\mathsf{DS})$.

From line 296 we know that $\mathsf{restricted}_u = \mathsf{false}$ when $\mathcal{A}_{\mathsf{DS}}$ aborts, so vk_u is the same verification key used by UNIQ. Since $\mathbf{ctable}_u[r_u + 1]$ was not $\perp$, we know $\mathsf{restricted}_{\overline{u}}$ was false when this ciphertext was sent. This ensures that the sequence of strings used to update the signing key before this ciphertext was sent is the same sequence just used for verification.

Claim 7, $\Pr[G_5] \leq \Pr[\text{INDEXP}^{\mathcal{A}_{\mathsf{PKE}}}_{\mathsf{PKE}}]$. At last we can reduce directly to the security of the public key encryption scheme PKE. To do so we build an adversary against its security which simulates the view of $\mathcal{D}$ and forwards all of $\mathcal{D}$'s valid challenge queries to its own encryption oracle. Here we use the phrase "valid challenge query" to refer to calls that $\mathcal{D}$ makes to SEND for which $m_0 \neq m_1$ and G_5 would not return $\perp$. Then the adversary will output $\mathcal{D}$'s guess at the secret bit as its own (except when $\mathcal{D}$ causes an early abort, in which case the adversary simply returns 1). Such an adversary $\mathcal{A}_{\mathsf{PKE}}$ is shown in Fig. 19.

The adversary $\mathcal{A}_{\mathsf{PKE}}$ keeps track of the different keys it requests from INDEXP by labelling them with tuples of the form $\Lambda = (s, u)$, where u is the user who creates the key pair while sending a message and s was the value of s_u at that time. When $\mathcal{D}$ makes expose queries, $\mathcal{A}_{\mathsf{PKE}}$ makes calls to its EXP oracle as necessary to expose the correct state. Whenever a new PKE key pair should be created, $\mathcal{A}_{\mathsf{PKE}}$ does so via a NEWUSER query unless the randomness for that key generation has been exposed, in which case it just generates the key pair by itself with said exposed randomness. Whenever $\mathcal{A}_{\mathsf{PKE}}$ needs the encryption of a message or decryption of a ciphertext, it will forward this on to the corresponding oracle in INDEXP except when it already has necessary information to perform this operation for itself.

Below we verify that adversary $\mathcal{A}_{\mathsf{PKE}}$ correctly simulates the view of $\mathcal{D}$ with the underlying bit of INDEXP playing the role of its secret bit. This gives $\Pr[G_5] \leq \Pr[\text{INDEXP}^{\mathcal{A}_{\mathsf{PKE}}}_{\mathsf{PKE}}]$, as desired.

Efficiency. We have shown the stated bound on the advantage of $\mathcal{D}$. The bounds on the number of oracle queries made by the adversaries and their runtimes can be verified by examining their code.

Correct simulation by $\mathcal{A}_{\mathsf{PKE}}$. We verify that none of the oracles called by $\mathcal{A}_{\mathsf{PKE}}$ will abort early and return $\perp$, except possibly DEC. We will argue that $\mathcal{A}_{\mathsf{PKE}}$ doesn't require the output of DEC to be non-$\perp$ in the cases when it returns $\perp$. Recall that $\mathcal{A}_{\mathsf{PKE}}$ uses user identifiers of the form $\Lambda = (s, u)$ for the key-pair created by u when sending their s-th ciphertext.

Queries to NEWUSER are made in SENDSIM only after s_u has been updated, so (since s_u is strictly increasing) such queries will never be made with $\mathbf{dk}[\Lambda] \neq \perp$. For other oracles, note that $\mathbf{dk}[(s, v)] = \perp$ will only hold if (i) $s > s_v$ or (ii) ($\mathsf{nextop} = (v, \text{"send"})$ or $\mathsf{restricted}_v$) was true when the s-th ciphertext was sent by v (this condition corresponding when $\mathcal{A}_{\mathsf{PKE}}$ simply samples the corresponding keys for itself). For purposes of analysis we can divide the oracle queries into two groups.

The first group is DEC, UPDDK, and EXP. For this group the oracle queries with $\Lambda = (s, v)$ are only made for $s \leq s_u$ so condition (i) will not hold. If condition (ii) held, then on line 317 $\mathbf{dk}_u[s]$ would have been set to a non-$\perp$ value so the checks before these three oracle queries would prevent the queries from being made. Note here that line 337 can set entries of $\mathbf{dk}_u$ back to $\perp$, but only

Adversary $\mathcal{A}_{\mathsf{PKE}}^{\textsc{NewUser},\textsc{UpdEk},\textsc{UpdDk},\textsc{Enc},\textsc{Dec},\textsc{Exp}}$

$r_{\mathcal{I}}^{rest} \leftarrow r_{\mathcal{R}}^{rest} \leftarrow \infty$; $s_{\mathcal{I}} \leftarrow r_{\mathcal{I}} \leftarrow s_{\mathcal{R}} \leftarrow r_{\mathcal{R}} \leftarrow 0$; $\mathsf{restricted}_{\mathcal{I}} \leftarrow \mathsf{restricted}_{\mathcal{R}} \leftarrow \mathsf{false}$
$\mathbf{forge}_{\mathcal{I}}[0 \ldots \infty] \leftarrow \mathbf{forge}_{\mathcal{R}}[0 \ldots \infty] \leftarrow$ "nontrivial" ; $\mathcal{X}_{\mathcal{I}} \leftarrow \mathcal{X}_{\mathcal{R}} \leftarrow 0$
$(sk_{\mathcal{I}}, vk_{\mathcal{R}}) \leftarrow\!\!\$\ \mathsf{DS.Kg}$; $(sk_{\mathcal{R}}, vk_{\mathcal{I}}) \leftarrow\!\!\$\ \mathsf{DS.Kg}$
$ek_{\mathcal{I}} \leftarrow \textsc{NewUser}((s_{\mathcal{R}}, \mathcal{R}))$; $ek_{\mathcal{R}} \leftarrow \textsc{NewUser}((s_{\mathcal{I}}, \mathcal{I}))$
$hk \leftarrow\!\!\$\ \mathsf{H.Kg}$; $\boldsymbol{\tau}_{r,\mathcal{I}}[0] \leftarrow \boldsymbol{\tau}_{r,\mathcal{R}}[0] \leftarrow \boldsymbol{\tau}_{s,\mathcal{I}}[0] \leftarrow \boldsymbol{\tau}_{s,\mathcal{R}}[0] \leftarrow \varepsilon$; $r_{\mathcal{I}}^{ack} \leftarrow r_{\mathcal{R}}^{ack} \leftarrow 0$
$(z_{\mathcal{I}}, z_{\mathcal{R}}) \leftarrow\!\!\$\ (\mathsf{Ch.SendRS})^2$; $(\eta_{\mathcal{I}}, \eta_{\mathcal{R}}) \leftarrow\!\!\$\ (\mathsf{Ch.RecvRS})^2$
$b' \leftarrow\!\!\$\ \mathcal{D}^{\textsc{SendSim},\textsc{RecvSim},\textsc{ExpSim}}$; Return b'

$\textsc{SendSim}(u, m_0, m_1, ad)$

If $\mathsf{nextop} \neq (u, \text{"send"})$ and $\mathsf{nextop} \neq \perp$ then return $\perp$
If $|m_0| \neq |m_1|$ then return $\perp$
If ($r_u < \mathcal{X}_u$ or $\mathsf{restricted}_u$ or $\mathbf{ch}_u[s_u + 1] =$ "forbidden") and $m_0 \neq m_1$ then
    Return $\perp$
$(z_1, z_2, z_3, z_4) \leftarrow z_u$; $s_u \leftarrow s_u + 1$; $(sk, vk) \leftarrow \mathsf{DS.Kg}(z_1)$
If $\mathsf{nextop} \neq (u, \text{"send"})$ and not $\mathsf{restricted}_u$ then $ek \leftarrow \textsc{NewUser}((s_u, u))$ else $(ek, \mathbf{dk}_u[s_u]) \leftarrow \mathsf{PKE.Kg}(z_2)$
$\ell \leftarrow (s_u, r_u, ad, vk, ek, \boldsymbol{\tau}_{r,u}[r_u], \boldsymbol{\tau}_{s,u}[s_u - 1])$; $\vec{\Delta}_e \leftarrow \boldsymbol{\tau}_{s,u}[r_u^{ack} + 1, \ldots, s_u - 1]$
If $m_0 \neq m_1$ then $c' \leftarrow \textsc{Enc}((r_u, \overline{u}), m_0, m_1, \ell)$ else $(ek', c') \leftarrow \mathsf{PKE.Enc}(ek_u, \ell, m_1, \vec{\Delta}_e; z_3)$
$v \leftarrow (c', \ell)$; $\sigma \leftarrow \mathsf{DS.Sign}(sk_u, v; z_4)$; $c \leftarrow (\sigma, v)$; $\boldsymbol{\tau}_{s,u}[s_u] \leftarrow \mathsf{H.Ev}(hk, c)$; $sk_u \leftarrow sk$
If $r_u \geq \mathcal{X}_u$ and not $\mathsf{restricted}_u$ then $\textsc{UpdEk}((r_u, \overline{u}), \boldsymbol{\tau}_{s,u}[s_u])$
$\mathsf{nextop} \leftarrow \perp$; $z_u \leftarrow\!\!\$\ \mathsf{Ch.SendRS}$
If not $\mathsf{restricted}_u$ and $\boldsymbol{\tau}_{s,u}[s_u] = \boldsymbol{\tau}_{r,\overline{u}}[s_u]$ then $\mathbf{abort}(1)$
If not $\mathsf{restricted}_u$ then $\mathbf{ctable}_{\overline{u}}[s_u] \leftarrow (c, ad)$
If $m_0 \neq m_1$ then $\mathbf{ch}_u[s_u] \leftarrow$ "done"
Return c

$\textsc{RecvSim}(u, c, ad)$

If $\mathsf{nextop} \neq (u, \text{"recv"})$ and $\mathsf{nextop} \neq \perp$ then return $\perp$
$(\eta_1, \eta_2) \leftarrow \eta_u$; $\mathsf{nextop} \leftarrow \perp$; $\eta_u \leftarrow\!\!\$\ \mathsf{Ch.RecvRS}$
$(\sigma, v) \leftarrow c$; $(c', \ell) \leftarrow v$; $(s', r', ad', vk', ek', \tau_r', \tau_s') \leftarrow \ell$
$(vk'', t) \leftarrow \mathsf{DS.Vrfy}(vk_u, \sigma, v, \boldsymbol{\tau}_s[r_u^{ack} + 1, \ldots, r'])$
If not t or $s' \neq r_u + 1$ or $\tau_r' \neq \boldsymbol{\tau}_{s,u}[r']$ or $\tau_s' \neq \boldsymbol{\tau}_{r,u}[r_u]$ or $ad' \neq ad$ then return $\perp$
If not $\mathsf{restricted}_u$ and $\mathbf{forge}_u[r_u + 1] \neq$ "trivial" and $(c, ad) \neq \mathbf{ctable}_u[r_u + 1]$ then $\mathbf{abort}(1)$
$((\sigma', v'), ad) \leftarrow \mathbf{ctable}_u[r_u + 1]$
If not $\mathsf{restricted}_u$ and $v = v'$ and $\sigma \neq \sigma'$ then $\mathbf{abort}(1)$
$r_u \leftarrow r_u + 1$; $r_u^{ack} \leftarrow r'$;
If $\mathbf{dk}_u[r_u^{ack}] = \perp$ then $m \leftarrow \textsc{Dec}((r_u^{ack}, u), c', \ell)$ else $m \leftarrow \mathsf{PKE.Dec}(\mathbf{dk}_u[r_u^{ack}], \ell, c')$
$\boldsymbol{\tau}_{s,u}[0, \ldots, r_u^{ack} - 1] \leftarrow \perp$; $\mathbf{dk}_u[0, \ldots, r_u^{ack} - 1] \leftarrow \perp$; $\boldsymbol{\tau}_{r,u}[r_u] \leftarrow \mathsf{H.Ev}(hk, c)$
$sk_u \leftarrow \mathsf{DS.UpdSk}(sk_u, \boldsymbol{\tau}_{r,u}[r_u]; \eta_1)$; $vk_u \leftarrow vk'$; $ek_u \leftarrow ek'$
If $r_u \geq \mathcal{X}_u$ and not $\mathsf{restricted}_u$ then for $i \in [r_u^{ack} + 1, s_u]$ do $\textsc{UpdEk}((r_u, \overline{u}), \boldsymbol{\tau}_{s,u}[i])$
For $i \in [r_u^{ack}, s_u]$ if $\mathbf{dk}_u[i] = \perp$ do $\textsc{UpdDk}((i, u), \boldsymbol{\tau}_{r,u}[r_u])$ else $\mathbf{dk}_u[i] \leftarrow \mathsf{PKE.UpdDk}(\mathbf{dk}_u[i], \boldsymbol{\tau}_{r,u}[r_u]; \eta_2^i)$
If $(c, ad) \neq \mathbf{ctable}_u[r_u]$ then $\mathsf{restricted}_u \leftarrow \mathsf{true}$; $r_u^{rest} \leftarrow \min\{r_u, r_u^{rest}\}$
If not $\mathsf{restricted}_{\overline{u}}$ and $\boldsymbol{\tau}_{r,u}[r_u^{rest}] = \boldsymbol{\tau}_{s,\overline{u}}[r_u^{rest}] \neq \perp$ then $\mathbf{abort}(1)$
If $\mathsf{restricted}_u$ then return m
Return $\perp$

$\textsc{ExpSim}(u, \mathsf{rand})$

If $\mathsf{nextop} \neq \perp$ then return $\perp$
If not $\mathsf{restricted}_u$ and $\exists i \in (r_u, s_{\overline{u}}]$ s.t. $\mathbf{ch}_{\overline{u}}[i] =$ "done" then
    Return $\perp$
For $i \in [r_u^{ack}, s_u]$ if $\mathbf{dk}_u[i] = \perp$ then $\mathbf{dk}_u[i] \leftarrow \textsc{Exp}((i, u))$
$st \leftarrow (s_u, r_u, sk_u, vk_u, ek_u, \mathbf{dk}_u, hk, \boldsymbol{\tau}_{r,u}[r_u], \boldsymbol{\tau}_{s,u})$
If $\mathsf{restricted}_u$ then return (st, z_u, η_u)
$\mathbf{forge}_{\overline{u}}[s_u + 1] \leftarrow$ "trivial" ; $(z, \eta) \leftarrow (\varepsilon, \varepsilon)$; $\mathcal{X}_{\overline{u}} \leftarrow s_u + 1$
If $\mathsf{rand} =$ "send" then
    $\mathsf{nextop} \leftarrow (u, \text{"send"})$; $z \leftarrow z_u$; $\mathcal{X}_{\overline{u}} \leftarrow s_u + 2$
    $\mathbf{forge}_{\overline{u}}[s_u + 2] \leftarrow$ "trivial" ; $\mathbf{ch}_u[s_u + 1] \leftarrow$ "forbidden"
Else if $\mathsf{rand} =$ "recv" then
    $\mathsf{nextop} \leftarrow (u, \text{"recv"})$; $\eta \leftarrow \eta_u$
Return (st, z, η)

Figure 19: Adversary $\mathcal{A}_{\mathsf{PKE}}$ attacking PKE. Highlighting indicates changes from $\mathrm{G}_4, \mathrm{G}_5$.

for entries with indices less than r_u^{ack} and these oracle queries are only made for indices at least as large as r_u^{ack} (which is a strictly increasing value).

The second group of oracles is Enc and UpdEk. Note that the latter can be called both by SendSim and by RecvSim. All of these queries are done with $\Lambda = (r_u, \overline{u})$ and only if $r_u \geq \mathcal{X}_u$ and $\mathsf{restricted}_u = \mathsf{false}$. (In the case of Enc this is ensured by the fact that $m_0 \neq m_1$ and line 314 must have evaluated to false.) That $\mathsf{restricted}_u = \mathsf{false}$ ensures that $r_u \leq s_{\overline{u}}$, so condition (i) cannot hold. For the latter condition note that nextop can only be set to ($\overline{u}$, "send") in Exp. But this would have set $\mathcal{X}_u$ such that $r_u \geq \mathcal{X}_u$ would be false. If $\mathsf{restricted}_{\overline{u}}$ held when the r_u-th ciphertext was sent by $\overline{u}$, then $\mathsf{restricted}_u$ would have been set when u received the corresponding ciphertext. So condition (ii) also cannot hold.

So none of the oracles will abort early due to checks involving $\mathbf{dk}$. It remains to individually analyze the possibilities of Enc, Dec, or Exp aborting early for other reasons.

It will never be the case that $|m_0| \neq |m_1|$ in Enc because that would have resulted in SendSim aborting early. So consider the possibility that a call to Enc is made when $\vec{\Delta}'[(r_u, \overline{u})] \sqsubseteq \vec{\Delta}_e[(r_u, \overline{u})]$. Note that $\vec{\Delta}_e[(r_u, \overline{u})] = \boldsymbol{\tau}_{s,u}[r_u^{ack} + 1, \ldots, s_u - 1]$ and $\vec{\Delta}'[(r_u.\overline{u})] = \boldsymbol{\tau}_{r,\overline{u}}[r_u^{ack} + 1, \ldots, j]$ for some j. The value of $\vec{\Delta}'[(r_u, \overline{u})]$ must have been set by a call to Exp by ExpSim when $r_{\overline{u}}$ was equal to j. If $\mathsf{restricted}_{\overline{u}}$ did not hold at the time, then $\mathcal{X}_u$ would have been set to a value greater than r_u so this Enc query would not be made. Also, $\mathsf{restricted}_u$ cannot hold when the Enc query is made. This implies that $\mathsf{restricted}_{\overline{u}}$ did not hold when the last ciphertext received by u was sent by $\overline{u}$. Thus, $r_u^{ack} < r_{\overline{u}}^{rest} \leq j$. Then $\vec{\Delta}'[(r_u, \overline{u})] \sqsubseteq \vec{\Delta}_e[(r_u, \overline{u})]$ would imply that $\boldsymbol{\tau}_{s,u}[r_{\overline{u}}^{rest}] = \boldsymbol{\tau}_{r,\overline{u}}[r_{\overline{u}}^{rest}]$ which is impossible. If $\boldsymbol{\tau}_{r,\overline{u}}[r_{\overline{u}}^{rest}]$ was the first of these set to a non-$\perp$ value, then when $\boldsymbol{\tau}_{s,u}[r_{\overline{u}}^{rest}]$ was set it would have caused $\mathcal{A}_{\mathsf{PKE}}$ to abort on line 323. If $\boldsymbol{\tau}_{s,u}[r_{\overline{u}}^{rest}]$ was set first, then when $\boldsymbol{\tau}_{r,\overline{u}}[r_{\overline{u}}^{rest}]$ was set it would have caused $\mathcal{A}_{\mathsf{PKE}}$ to abort on line 342.

Consider the possibility that a call to Dec is made when $(\vec{\Delta}_d[(r_u^{ack}, u)], c', \ell) \in S[(r_u^{ack}, u)]$. Then there was a prior query of the form $\textsc{Enc}((r_u^{ack}, u), \cdot, \cdot, \ell)$ which returned c'. By logic we have used previously (using lines 319 and 314), $\mathsf{restricted}_{\overline{u}} = \mathsf{false}$ must have held at the time. Then $\mathbf{ctable}_u[r_u]$ was set on line 324. If $\mathsf{restricted}_u = \mathsf{false}$ and $(c, ad) = \mathbf{ctable}_u[r_u]$, then $\mathsf{restricted}_u$ will still be false at the end of RecvSim, so $\perp$ will be returned to $\mathcal{D}$ no matter what Dec outputs. If $\mathsf{restricted}_u = \mathsf{false}$ and $(c, ad) \neq \mathbf{ctable}_u[r_u]$, then from line 334 we can see that $v \neq v'$ which is a contradiction because both are equal to (c', ℓ). So suppose that $\mathsf{restricted}_u$ holds. Note that $\vec{\Delta}_d[(r_u^{ack}, u)] = \boldsymbol{\tau}_{r,u}[j, \ldots, r_u - 1]$ where j is one more than the value r_u held when the the call $\textsc{NewUser}((r_u^{ack}, u))$ was made. Furthermore, $\vec{\Delta}_d[(r_u^{ack}, u)] = \boldsymbol{\tau}_{s,\overline{u}}[j, \ldots, r_u - 1]$ because it is equal to the value $\vec{\Delta}_e[(r_u^{ack}, u)]$ held during the relevant Enc query. For Dec to have been queried it must hold that $\mathbf{dk}_u[r_u^{ack}] = \perp$, so $\mathsf{restricted}_u$ did not hold when $\textsc{NewUser}((r_u^{ack}, u))$ was queried. This means $r_u^{rest} \in [j, \ldots, r_u - 1]$ so we have $\boldsymbol{\tau}_{r,u}[r_u^{rest}] = \boldsymbol{\tau}_{s,\overline{u}}[r_u^{rest}]$. This is impossible by the same reasoning used in the prior paragraph.

Consider the possibility that a call to Exp is made when $\exists(\vec{\Delta}, c, \ell) \in S[(i, u)]$ such that $\vec{\Delta}_d[(i, u)] \sqsubseteq \vec{\Delta}$. Note that $\vec{\Delta}$ is equal to the value of $\vec{\Delta}_e[(i, u)]$ during some previous Enc query. By prior logic, $\mathsf{restricted}_{\overline{u}} = \mathsf{false}$ must have held at the time. Suppose $\mathsf{restricted}_u = \mathsf{false}$. By 346, it must hold that $\mathbf{ch}_{\overline{u}}[j] \neq$ "done" for all $j \in (r_u, s_{\overline{u}}]$ and note that $\mathbf{ch}_{\overline{u}}[s] =$ "done". Let s denote the value of $s_{\overline{u}}$ when the relevant Enc query was made. Note that $s \leq s_{\overline{u}}$ because $s_{\overline{u}}$ never decreases and $s > r_u$ because $|\vec{\Delta}_d[(i, u)]| \leq |\vec{\Delta}_e[(i, u)]|$. This is a contradiction. So suppose that $\mathsf{restricted}_u$ holds. Note that $\vec{\Delta}_d[(i, u)] = \boldsymbol{\tau}_{r,u}[j, \ldots, r_u]$ where j is one more than the value r_u held when the the call $\textsc{NewUser}((r_u^{ack}, u))$ was made. Furthermore, $\vec{\Delta}_d[(r_u^{ack}, u)] = \boldsymbol{\tau}_{s,\overline{u}}[j, \ldots, r_u]$ because it is equal to

the prefix of $\vec{\Delta}_e[(i,u)]$ held during the relevant ENC query. For DEC to have been queried it must hold that $\mathbf{dk}_u[i] = \perp$, so $\mathsf{restricted}_u$ did not hold when NEWUSER$((i,u))$ was queried. This means $r_u^{rest} \in [j, \ldots, r_u]$ so we have $\boldsymbol{\tau}_{r,u}[r_u^{rest}] = \boldsymbol{\tau}_{s,\overline{u}}[r_u^{rest}]$. This is impossible by the same reasoning used two paragraphs prior. ▌

Acknowledgments

We thank Mihir Bellare for extensive discussion on preliminary versions of this paper. We thank the CRYPTO 2018 reviewers for their comments.

References

[1] M. Bellare, A. Boldyreva, and S. Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In B. Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 259–274, Bruges, Belgium, May 14–18, 2000. Springer, Heidelberg, Germany. 7

[2] M. Bellare, A. Desai, E. Jokipii, and P. Rogaway. A concrete security treatment of symmetric encryption. In *38th FOCS*, pages 394–403, Miami Beach, Florida, Oct. 19–22, 1997. IEEE Computer Society Press. 3

[3] M. Bellare, A. Desai, D. Pointcheval, and P. Rogaway. Relations among notions of security for public-key encryption schemes. In H. Krawczyk, editor, *CRYPTO'98*, volume 1462 of *LNCS*, pages 26–45, Santa Barbara, CA, USA, Aug. 23–27, 1998. Springer, Heidelberg, Germany. 11

[4] M. Bellare, T. Kohno, and C. Namprempre. Breaking and provably repairing the ssh authenticated encryption scheme: A case study of the encode-then-encrypt-and-mac paradigm. *ACM Transactions on Information and System Security (TISSEC)*, 7(2):206–241, 2004. 2

[5] M. Bellare and S. K. Miner. A forward-secure digital signature scheme. In M. J. Wiener, editor, *CRYPTO'99*, volume 1666 of *LNCS*, pages 431–448, Santa Barbara, CA, USA, Aug. 15–19, 1999. Springer, Heidelberg, Germany. 4, 5, 10

[6] M. Bellare, A. O'Neill, and I. Stepanovs. Forward-security under continual leakage. Cryptology ePrint Archive, Report 2017/476, 2017. `http://eprint.iacr.org/2017/476`. 44

[7] M. Bellare, T. Ristenpart, and S. Tessaro. Multi-instance security and its application to password-based cryptography. In R. Safavi-Naini and R. Canetti, editors, *CRYPTO 2012*, volume 7417 of *LNCS*, pages 312–329, Santa Barbara, CA, USA, Aug. 19–23, 2012. Springer, Heidelberg, Germany. 42

[8] M. Bellare and P. Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In S. Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Heidelberg, Germany. 6, 29

[9] M. Bellare, A. C. Singh, J. Jaeger, M. Nyayapati, and I. Stepanovs. Ratcheted encryption and key exchange: The security of messaging. In J. Katz and H. Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 619–650, Santa Barbara, CA, USA, Aug. 20–24, 2017. Springer, Heidelberg, Germany. 3, 5, 12, 13, 19, 42, 43

[10] M. Bellare and B. Yee. Forward-security in private-key cryptography. In M. Joye, editor, *Topics in Cryptology — CT-RSA 2003*, pages 1–18, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. 5

[11] A. Boldyreva, J. P. Degabriele, K. G. Paterson, and M. Stam. Security of symmetric encryption in the presence of ciphertext fragmentation. In D. Pointcheval and T. Johansson, editors, *EUROCRYPT 2012*, volume 7237 of *LNCS*, pages 682–699, Cambridge, UK, Apr. 15–19, 2012. Springer, Heidelberg, Germany. 4

[12] N. Borisov, I. Goldberg, and E. Brewer. Off-the-record communication, or, why not to use pgp. In *Proceedings of the 2004 ACM Workshop on Privacy in the Electronic Society*, WPES '04, pages 77–84, New York, NY, USA, 2004. ACM. 3

[13] R. Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145, Las Vegas, NV, USA, Oct. 14–17, 2001. IEEE Computer Society Press. 2

[14] R. Canetti, S. Halevi, and J. Katz. A forward-secure public-key encryption scheme. *Journal of Cryptology*, 20(3):265–294, July 2007. 5

[15] R. Canetti and H. Krawczyk. Analysis of key-exchange protocols and their use for building secure channels. In B. Pfitzmann, editor, *EUROCRYPT 2001*, volume 2045 of *LNCS*, pages 453–474, Innsbruck, Austria, May 6–10, 2001. Springer, Heidelberg, Germany. 2

[16] K. Cohn-Gordon, C. Cremers, B. Dowling, L. Garratt, and D. Stebila. A formal security analysis of the Signal messaging protocol. In *Proc. IEEE European Symposium on Security and Privacy (EuroS&P) 2017*. IEEE, April 2017. To appear. 3, 19, 42

[17] K. Cohn-Gordon, C. Cremers, and L. Garratt. On post-compromise security. In *2016 IEEE 29th Computer Security Foundations Symposium (CSF)*, pages 164–178, June 2016. 2

[18] Y. Desmedt and Y. Frankel. Threshold cryptosystems. In G. Brassard, editor, *CRYPTO'89*, volume 435 of *LNCS*, pages 307–315, Santa Barbara, CA, USA, Aug. 20–24, 1990. Springer, Heidelberg, Germany. 4

[19] W. Diffie, P. C. van Oorschot, and M. J. Wiener. Authentication and authenticated key exchanges. *Designs, Codes and Cryptography*, 2(2):107–125, June 1992. 4, 5

[20] Y. Dodis, J. Katz, S. Xu, and M. Yung. Key-insulated public key cryptosystems. In L. R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 65–82, Amsterdam, The Netherlands, Apr. 28 – May 2, 2002. Springer, Heidelberg, Germany. 4

[21] Y. Dodis, J. Katz, S. Xu, and M. Yung. Strong key-insulated signature schemes. In Y. Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 130–144, Miami, FL, USA, Jan. 6–8, 2003. Springer, Heidelberg, Germany. 4

[22] Y. Dodis, W. Luo, S. Xu, and M. Yung. Key-insulated symmetric key cryptography and mitigating attacks against cryptographic cloud software. In H. Y. Youm and Y. Won, editors, *ASIACCS 12*, pages 57–58, Seoul, Korea, May 2–4, 2012. ACM Press. 4

[23] T. P. (editor) and M. Marlinspike. The double ratchet algorithm. `https://whispersystems.org/docs/specifications/doubleratchet/`, Nov. 20, 2016. Accessed: 2017-06-03. 3, 43

[24] M. Fischlin, F. Günther, G. A. Marson, and K. G. Paterson. Data is a stream: Security of stream-based channels. In R. Gennaro and M. J. B. Robshaw, editors, *CRYPTO 2015, Part II*, volume 9216 of *LNCS*, pages 545–564, Santa Barbara, CA, USA, Aug. 16–20, 2015. Springer, Heidelberg, Germany. 4

[25] C. Gentry and A. Silverberg. Hierarchical ID-based cryptography. In Y. Zheng, editor, *ASIACRYPT 2002*, volume 2501 of *LNCS*, pages 548–566, Queenstown, New Zealand, Dec. 1–5, 2002. Springer, Heidelberg, Germany. 4, 47

[26] M. D. Green and I. Miers. Forward secure asynchronous messaging from puncturable encryption. In *2015 IEEE Symposium on Security and Privacy*, pages 305–320, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press. 5

[27] C. G. Günther. An identity-based key-exchange protocol. In J.-J. Quisquater and J. Vandewalle, editors, *EUROCRYPT'89*, volume 434 of *LNCS*, pages 29–37, Houthalen, Belgium, Apr. 10–13, 1990. Springer, Heidelberg, Germany. 4, 5

[28] F. Günther and S. Mazaheri. A formal treatment of multi-key channels. In J. Katz and H. Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 587–618, Santa Barbara, CA, USA, Aug. 20–24, 2017. Springer, Heidelberg, Germany. 3, 19, 42

[29] H. Krawczyk. The order of encryption and authentication for protecting communications (or: How secure is SSL?). In J. Kilian, editor, *CRYPTO 2001*, volume 2139 of *LNCS*, pages 310–331, Santa Barbara, CA, USA, Aug. 19–23, 2001. Springer, Heidelberg, Germany. 2

[30] A. Langley. Pond. GitHub repository, README.md, `https://github.com/agl/pond/commit/7bb06244b9aa121d367a6d556867992d1481f0c8`, 2012. Accessed: 2017-06-03. 3

[31] G. A. Marson and B. Poettering. Security notions for bidirectional channels. *IACR Trans. Symm. Cryptol.*, 2017(1):405–426, 2017. 2, 3, 11, 12, 13, 15, 43

[32] M. Mignotte. How to share a secret? In T. Beth, editor, *EUROCRYPT'82*, volume 149 of *LNCS*, pages 371–375, Burg Feuerstein, Germany, Mar. 29 – Apr. 2, 1983. Springer, Heidelberg, Germany. 4

[33] C. Namprempre. Secure channels based on authenticated encryption schemes: A simple characterization. In Y. Zheng, editor, *ASIACRYPT 2002*, volume 2501 of *LNCS*, pages 515–532, Queenstown, New Zealand, Dec. 1–5, 2002. Springer, Heidelberg, Germany. 2

[34] Open Whisper Systems. Signal protocol library for java/android. GitHub repository, `https://github.com/WhisperSystems/libsignal-protocol-java`, 2017. Accessed: 2017-06-03. 2

[35] R. Ostrovsky and M. Yung. How to withstand mobile virus attacks (extended abstract). In L. Logrippo, editor, *10th ACM PODC*, pages 51–59, Montreal, Quebec, Canada, Aug. 19–21, 1991. ACM. 4

[36] B. Poettering and P. Rösler. Ratcheted key exchange, revisited. Cryptology ePrint Archive, Report 2018/296, 2018. `https://eprint.iacr.org/2018/296`. 5

[37] P. Rogaway. Authenticated-encryption with associated-data. In V. Atluri, editor, *ACM CCS 02*, pages 98–107, Washington D.C., USA, Nov. 18–22, 2002. ACM Press. 2

[38] P. Rogaway and T. Shrimpton. A provable-security treatment of the key-wrap problem. In S. Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 373–390, St. Petersburg, Russia, May 28 – June 1, 2006. Springer, Heidelberg, Germany. 3, 15

[39] A. Shamir. How to share a secret. *Communications of the Association for Computing Machinery*, 22(11):612–613, Nov. 1979. 4

[40] V. Shoup. On formal models for secure key exchange. Cryptology ePrint Archive, Report 1999/012, 1999. `http://eprint.iacr.org/1999/012`. 2

[41] V. Shoup. A proposal for an iso standard for public key encryption. Cryptology ePrint Archive, Report 2001/112, 2001. `https://eprint.iacr.org/2001/112`. 10

[42] M. Tompa and H. Woll. How to share a secret with cheaters. *Journal of Cryptology*, 1(2):133–138, 1988. 4

[43] N. Unger, S. Dechand, J. Bonneau, S. Fahl, H. Perl, I. Goldberg, and M. Smith. SoK: Secure messaging. In *2015 IEEE Symposium on Security and Privacy*, pages 232–249, San Jose, CA, USA, May 17–21, 2015. IEEE Computer Society Press. 2

[44] WhatsApp Blog. Connecting one billion users every day. `https://blog.whatsapp.com/10000631/Connecting-One-Billion-Users-Every-Day`, July 26, 2017. Accessed: 2018-02-12. 2

A Comparison to recent definitions

Three recents works we studied while deciding how to write our security definition were the works of CCDGS [16], BCJNS [9], and GM [28]. While they all ultimately were interested in different settings (and CCDGS did not even model any form of encryption) they all share the commonality of modeling security in a setting where there are different "stages" of keys. All three works made distinct decisions in how to address challenges in different stages, so its worth discussing the repercussions of these different decisions.

In BCJNS and GM a stage corresponds to a counter i representing how many times the relevant key has been updated. In CCDGS a stage corresponds to a tuple (u, j, i) corresponding to user u in step i of its j-th protocol execution. In our work a stage could correspond to a tuple (u, O, s, r) corresponding to a query to user $u \in \{\mathcal{I}, \mathcal{R}\}$ with oracle $\mathrm{O} \in \{\textsc{Send}, \textsc{Recv}\}$ when $s_u = s$ and $r_u = r$.

CCDGS chose to only allow the adversary to make a challenge query in a single stage for which there is a single corresponding bit that it must guess. BCJNS also has only a single challenge bit, but allows the adversary to make challenge queries in arbitrarily many stages all of which share that bit. GM also allows challenge queries in arbitrarily many stages, but samples a separate challenge bit for each stage. At the end of the game the adversary outputs both a bit and the index of the stage for which it is trying to guess the bit. BCJNS thus needed to keep track throughout of whether an adversary has "cheated" by doing something it is not allowed to which would tell it was the secret bit was. CCDGS and GM only need to perform this check for the particular stage for which the adversary attempts to guess the bit. This check is still somewhat global because actions in other stages may affect whether the adversary has cheated for the challenge stage. We will refer to these different styles of definitional choices as CCDGS, BCJNS, or GM security and discuss them broadly without regard to how stages are defined or what the underlying game is.

Qualitatively these three definitional choices result in equivalent definitions. BCJNS security and GM security can both easily be shown to imply CCDGS security. In the other direction, a hybrid argument can be used to show that CCDGS security implies BCJNS security and an index guessing proof can be used to show that it implies GM security. However, both of these proof techniques introduce a factor q loss of security (where q is the maximum number of stages an adversary ever interacts with) so CCDGS security appears to be quantitively weaker than the other two. BCJNS and GM security appear to be quantitatively incomparable; the only way we are aware of to show that one implies the other is to use CCDGS security as an intermediate step which introduces a factor q loss of security in *both* directions.

BCJNS security challenges the adversary to learn one bit corresponding exactly to which of two possible "worlds" it exists in. GM security instead challenges the adversary to learn one bit about the exponential number of "worlds" it may exist in (though, to be clear, it can choose to cheat to learn all of the bits of information about which "world" it is in other than the one bit it attempts to guess). BCJNS show that their style of definition allows tight security reductions from multi-user security assumptions about the underlying primitives. GM did not aim to give tight security notions, but it is likely that tight reductions could be given from a variant multi-user style assumption in which the game samples an independent challenge bit for every user and the adversary wins if it can guess the bit corresponding to a single user. We are not aware of any works that use such a multi-user definition directly and do not know if techniques used to tightly prove security for standard multi-user definition would extend to such a definition.

This multi-user style definition is similar to multi-instance style definitions as introduced by [7]

which sample an independent bit for each user and require the adversary guess the xor of *all* the challenge bits. They give away the underlying challenge bit when the secrets of a user are exposed, so considering adversaries who expose all users except one gives a notion essentially equivalent to the multi-user notion sketched above.

When considering the above, we ultimately decided to follow a BCJNS style of security definition and provide tight reductions from the standard multi-user security of underlying primitives.

B Security implications of correctness notions

The correctness of a channel will have security implications outside of the scope of what we capture in our coming formalism. We suggest that CORR correctness may be a more appropriate notion of correctness in the secure messaging setting, but also note some plausible scenarios that could be applicable in the secure messaging setting for which CORR$\perp$ is more appropriate.

Recall that the correctness required by CORR$\perp$ is identical to that of Marson and Poettering [31][4] and is the standard notion for channels. It follows a common convention that a user will permanently refuse to send or receive future ciphertexts once they have received an invalid ciphertext. In practice this then requires that the connection be re-established for communication to continue. In secure messaging, securely re-establishing a connection is typically quite costly because it requires out-of-band human interaction. The CORR notion of correctness instead uses a form of robustness (analogous to that of [9]) to avoid that requirement. Thus this correctness prevents a denial-of-service attack by which an adversary without any secret knowledge can send a single incorrect ciphertext to either party to kill the communication channel.

Against a stronger adversary the correctness required by CORR$\perp$ prevents a worse attack. Suppose an attacker can compromise the state of user u. Then it can impersonate u and send messages to $\overline{u}$. Under CORR$\perp$ correctness the next time u attempts to send a message to $\overline{u}$ if the attacker cannot block this communication then this will permanently kill the channel and the users will know something has gone wrong. Under CORR correctness, this extra communication would simply be silently dropped. The users would need to communicate out-of-band to realize something had gone wrong (perhaps when u notices that $\overline{u}$ has stopped replying to any of its messages). For this reason an application using CORR correct should likely warn the user when it silently drops a message.

If the attacker even more powerful and able to block the outgoing communication sent by u from reaching $\overline{u}$ then the distinction between the two notions goes away. The users have to again fall back on out-of-band communication of the compromised status when u realizes that $\overline{u}$ is not receiving its messages.

The specification of the Double Ratchet Algorithm states *"If an exception is raised (e.g. message authentication failure) then the message is discarded and changes to the state object are discarded."* [23, Section 3.5]. This indicates that it would satisfy CORR. However, the Double Ratchet Algorithm was clearly designed to achieve a stricter notion of correctness because it intentionally accepts ciphertexts *in any order*. This goes against the standard security requirement that ciphertexts should only be accepted in the order they were sent, so we intentionally do not provide a correctness notion that captures this.

[4]This holds when requiring perfect security, which both works do. The two notions would be distinct if we considered computational correctness.

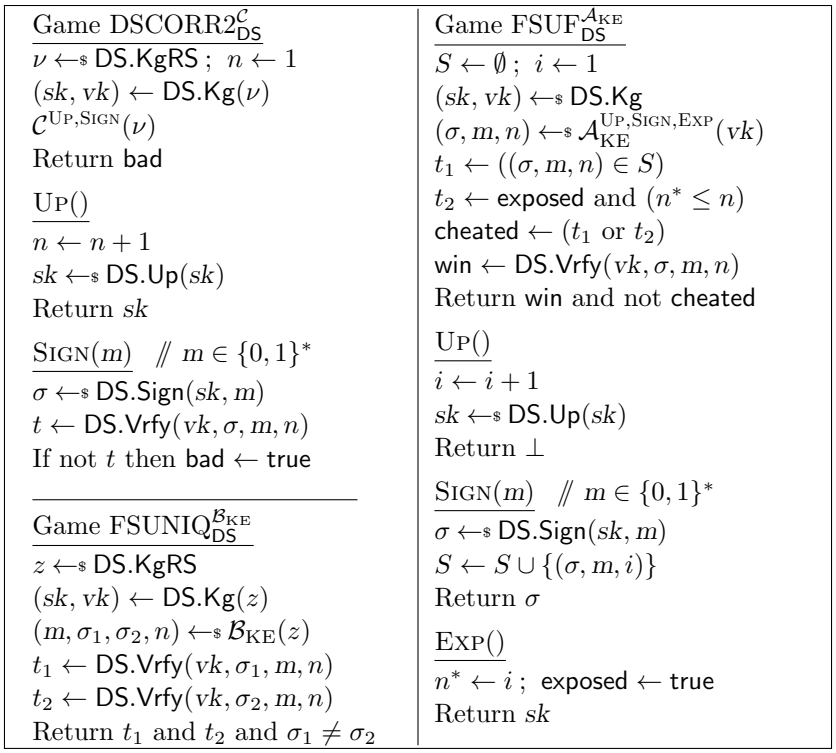

Figure 20: Games defining correctness, uniqueness, and forward security of key-evolving digital signature scheme DS.

C Construction of key-updatable digital signatures

In this section we formally specify the construction of a key-updatable digital signature scheme that we sketched in Section 3.1. For this purpose we use a key-evolving digital signature scheme.

Key-evolving digital signature schemes. A key-evolving digital signature scheme is a digital signature scheme with an additional algorithm DS.Up and with a modified verification algorithm DS.Vrfy. Update algorithm DS.Up takes a signing key sk to return an updated signing key, denoted by $sk \leftarrow\!\!\$\ \mathsf{DS.Up}(sk)$. Modified verification algorithm DS.Vrfy takes verification key vk, signature σ, message m, and time period $n \in \mathbb{N}$ to return a decision $t \in \{\mathsf{true}, \mathsf{false}\}$ regarding whether σ is a valid signature of m under vk for the n-th secret key, denoted by $t \leftarrow \mathsf{DS.Vrfy}(vk, \sigma, m, n)$.

Correctness is defined by game DSCORR2 in Fig. 20. For adversary $\mathcal{C}$ we define $\mathsf{Adv}^{\mathsf{dscorr2}}_{\mathsf{DS}}(\mathcal{C}) = \Pr[\text{DSCORR2}^{\mathcal{C}}_{\mathsf{DS}}]$ and require that $\mathsf{Adv}^{\mathsf{dscorr2}}_{\mathsf{DS}}(\mathcal{C}) = 0$ for all (even unbounded) adversaries.

Forward-secure signatures. Forward security of a key-evolving digital signature scheme asks that, even if the key is exposed in some time period n^*, it should be computationally hard to forge a valid signature for any prior time period $n < n^*$. Our definition follows that of [6]. Formally, consider game FSUF shown in Fig. 20, associated to a key-evolving digital signature scheme DS and an adversary $\mathcal{A}_{\text{KE}}$. The game generates a digital signature key pair for the initial time period $i = 1$ and runs the adversary with the verification key as input. The goal of the adversary is to forge a signature for an arbitrary time period n. The adversary is provided with access to oracles UP, SIGN and EXP. Oracle UP is used to advance into the next time period, incrementing its index i and updating the secret key of the scheme accordingly. Oracle SIGN uses the current

Algorithm $\mathsf{DS}_{\mathrm{KU}}.\mathsf{Kg}$
$(sk_{\mathrm{KE}}, vk_{\mathrm{KE}}) \leftarrow\!\!\$\ \mathsf{DS}_{\mathrm{KE}}.\mathsf{Kg}$
$\boldsymbol{\Sigma}[1,\ldots,\infty] \leftarrow \varepsilon\ ;\ \vec{\Delta}[1,\ldots,\infty] \leftarrow \varepsilon$
$sk \leftarrow (sk_{\mathrm{KE}}, 1, \boldsymbol{\Sigma})\ ;\ vk \leftarrow (vk_{\mathrm{KE}}, 1, \vec{\Delta})$
Return (sk, vk)

Algorithm $\mathsf{DS}_{\mathrm{KU}}.\mathsf{Vrfy}(vk, \sigma, m)$
$(vk_{\mathrm{KE}}, i_{\max}, \vec{\Delta}) \leftarrow vk$
$(\sigma_m, i, \boldsymbol{\Sigma}) \leftarrow \sigma$
If $i \neq i_{\max}$ then return false
$t \leftarrow \mathsf{DS}_{\mathrm{KE}}.\mathsf{Vrfy}(vk_{\mathrm{KE}}, \sigma_m, 1 \,\|\, m, i)$
For $j = 1, \ldots, (i-1)$ do
  $t \leftarrow t$ and $\mathsf{DS}_{\mathrm{KE}}.\mathsf{Vrfy}(vk_{\mathrm{KE}}, \boldsymbol{\Sigma}[j], 0 \,\|\, \vec{\Delta}[j], j)$
Return t

Algorithm $\mathsf{DS}_{\mathrm{KU}}.\mathsf{Sign}(sk, m)$
$(sk_{\mathrm{KE}}, i, \boldsymbol{\Sigma}) \leftarrow sk$
$\sigma_m \leftarrow\!\!\$\ \mathsf{DS}_{\mathrm{KE}}.\mathsf{Sign}(sk_{\mathrm{KE}}, 1 \,\|\, m)$
$\sigma \leftarrow (\sigma_m, i, \boldsymbol{\Sigma})$
Return σ

Algorithm $\mathsf{DS}_{\mathrm{KU}}.\mathsf{UpdSk}(sk, \Delta)$
$(sk_{\mathrm{KE}}, i, \boldsymbol{\Sigma}) \leftarrow sk$
$\boldsymbol{\Sigma}[i] \leftarrow\!\!\$\ \mathsf{DS}_{\mathrm{KE}}.\mathsf{Sign}(sk_{\mathrm{KE}}, 0 \,\|\, \Delta)$
$sk_{\mathrm{KE}} \leftarrow\!\!\$\ \mathsf{DS}_{\mathrm{KE}}.\mathsf{Up}(sk_{\mathrm{KE}})$
$sk \leftarrow (sk_{\mathrm{KE}}, i+1, \boldsymbol{\Sigma})$
Return sk

Algorithm $\mathsf{DS}_{\mathrm{KU}}.\mathsf{UpdVk}(vk, \Delta)$
$(vk_{\mathrm{KE}}, i, \vec{\Delta}) \leftarrow vk\ ;\ \vec{\Delta}[i] \leftarrow \Delta$
$vk \leftarrow (vk_{\mathrm{KE}}, i+1, \vec{\Delta})$
Return vk

Figure 21: Key-updatable digital signature scheme $\mathsf{DS}_{\mathrm{KU}} = \mathsf{DS\text{-}CONS}[\mathsf{DS}_{\mathrm{KE}}]$.

signing key to return a signature σ for an arbitrary message m (for current time period i); note that adversary is not allowed to win the game by returning (σ, m, i) as its forgery. Finally, oracle Exp exposes the current signing key. Without loss of generality, the adversary can only call this oracle once. If the exposure happened in time period n^*, then the adversary is only considered to win the game if it returns a forgery for some time period $n < n^*$, meaning it cannot use the exposed key to trivially win the game. The advantage of $\mathcal{A}_{\mathrm{KE}}$ in breaking the FSUF security of DS is $\mathsf{Adv}^{\mathsf{fsuf}}_{\mathsf{DS}}(\mathcal{A}_{\mathrm{KE}}) = \Pr[\mathrm{FSUF}^{\mathcal{A}_{\mathrm{KE}}}_{\mathsf{DS}}]$.

Signature uniqueness. Uniqueness of a key-evolving digital signature scheme requires that an adversary cannot find two distinct signatures that verify for the same message and the same round n. Consider the game FSUNIQ shown in Fig. 20, associated to a key-evolving signature scheme DS and an adversary $\mathcal{B}_{\mathrm{KE}}$. The advantage of $\mathcal{B}_{\mathrm{KE}}$ in breaking the FSUNIQ security of DS is $\mathsf{Adv}^{\mathsf{fsuniq}}_{\mathsf{DS}}(\mathcal{B}_{\mathrm{KE}}) = \Pr[\mathrm{FSUNIQ}^{\mathcal{B}_{\mathrm{KE}}}_{\mathsf{DS}}]$.

Key-updatable digital signature scheme $\mathsf{DS\text{-}CONS}[\mathsf{DS}_{\mathrm{KE}}]$. Let $\mathsf{DS}_{\mathrm{KE}}$ be a key-evolving digital signature scheme. We build a key-updatable digital signature scheme $\mathsf{DS}_{\mathrm{KU}} = \mathsf{DS\text{-}CONS}[\mathsf{DS}_{\mathrm{KE}}]$ as defined in Fig. 21, where $\mathsf{DS}_{\mathrm{KU}}.\mathsf{KgRS} = \mathsf{DS}_{\mathrm{KE}}.\mathsf{KgRS}$ and $\mathsf{DS}_{\mathrm{KU}}.\mathsf{SignRS} = \mathsf{DS}_{\mathrm{KE}}.\mathsf{SignRS}$.

The key-updatable digital signature scheme $\mathsf{DS}_{\mathrm{KU}}$ utilizes a key-evolving signature scheme $\mathsf{DS}_{\mathrm{KE}}$ in the following way. The signing key sk of $\mathsf{DS}_{\mathrm{KU}}$ consists of the corresponding signing key sk_{KE} of $\mathsf{DS}_{\mathrm{KE}}$, along with an empty list $\boldsymbol{\Sigma}$. In order to update the signing key sk of $\mathsf{DS}_{\mathrm{KU}}$ with a sequence of labels $\Delta_1, \Delta_2, \ldots, \Delta_q$, the scheme consecutively signs $\sigma_j \leftarrow\!\!\$\ \mathsf{DS}_{\mathrm{KE}}.\mathsf{Sign}(sk_{\mathrm{KE}}, 0 \,\|\, \Delta_j)$ with the underlying signing key sk_{KE} of $\mathsf{DS}_{\mathrm{KE}}$ and updates sk_{KE} using algorithm $\mathsf{DS}_{\mathrm{KE}}.\mathsf{Up}$, for each $j = 1, \ldots, q$. The resulting signing key sk of scheme $\mathsf{DS}_{\mathrm{KU}}$ is defined to contain the derived signing key sk_{KE} of scheme $\mathsf{DS}_{\mathrm{KE}}$ along with the produced signatures $\boldsymbol{\Sigma} = (\sigma_1, \sigma_2, \ldots)$ as per above.

The verification key vk of $\mathsf{DS}_{\mathrm{KU}}$ initially contains the corresponding verification key vk_{KE} of scheme $\mathsf{DS}_{\mathrm{KE}}$, along with an empty list $\vec{\Delta}$. To update vk with Δ, the key update Δ is appended to $\vec{\Delta}$.

The signature of a message m for key $sk = (sk_{\mathrm{KE}}, \boldsymbol{\Sigma})$ is defined as $(\sigma_m, \boldsymbol{\Sigma})$ for $\sigma_m \leftarrow\!\!\$$

$\mathsf{DS}_{\mathrm{KE}}.\mathsf{Sign}(sk_{\mathrm{KE}}, 1 \,\|\, m)$. Let $\mathbf{\Sigma} = (\sigma_1, \sigma_2, \ldots \sigma_{\kappa-1})$ and let $\vec{\Delta} = (\Delta_1, \Delta_2, \ldots, \Delta_{\kappa-1})$ for some κ. To verify the $\mathsf{DS}_{\mathrm{KU}}$ signature $(\sigma_m, \mathbf{\Sigma})$ against message m for verification key $vk = (vk_{\mathrm{KE}}, \vec{\Delta})$, one has to check that σ_i is a valid $\mathsf{DS}_{\mathrm{KE}}$ signature for message Δ_i for time period i, for each $i = 1, \ldots, \kappa-1$, and that σ_m is a valid signature for m in time period κ.

Security of DS-CONS[$\mathsf{DS}_{\mathrm{KE}}$]. Consider $\mathsf{DS}_{\mathrm{KU}} = \mathsf{DS\text{-}CONS}[\mathsf{DS}_{\mathrm{KE}}]$ for any key-evolving digital signature scheme $\mathsf{DS}_{\mathrm{KE}}$. We claim that for any adversary attacking UNIQ (signature uniqueness) or UFEXP (unforgeability under exposures) of $\mathsf{DS}_{\mathrm{KU}}$, there is an adversary that breaks FSUNIQ (signature uniqueness) or FSUF (forward security) of $\mathsf{DS}_{\mathrm{KE}}$, respectively, using roughly the same number of oracle queries and with roughly the same time efficiency.

Theorem C.1 *Let $\mathsf{DS}_{\mathrm{KE}}$ be a key-evolving digital signature scheme. Let $\mathsf{DS}_{\mathrm{KU}} = \mathsf{DS\text{-}CONS}[\mathsf{DS}_{\mathrm{KE}}]$. Let $\mathcal{B}_{\mathsf{DS}}$ be an adversary making at most 1 query to its* NewUser *oracle. Then we can build adversary $\mathcal{B}_{\mathrm{KE}}$ such that*

$$\mathsf{Adv}^{\mathsf{uniq}}_{\mathsf{DS}_{\mathrm{KU}}}(\mathcal{B}_{\mathsf{DS}}) \leq \mathsf{Adv}^{\mathsf{fsuniq}}_{\mathsf{DS}_{\mathrm{KE}}}(\mathcal{B}_{\mathrm{KE}}). \tag{2}$$

The runtime of $\mathcal{B}_{\mathrm{KE}}$ is about that of $\mathcal{B}_{\mathsf{DS}}$.

Theorem C.2 *Let $\mathsf{DS}_{\mathrm{KE}}$ be a key-evolving digital signature scheme. Let $\mathsf{DS}_{\mathrm{KU}} = \mathsf{DS\text{-}CONS}[\mathsf{DS}_{\mathrm{KE}}]$. Let $\mathcal{A}_{\mathsf{DS}}$ be an adversary making at most 1 query to its* NewUser *oracle, q_{UPD} queries to its* Upd *oracle, 1 query to its* Sign *oracle, and 1 queries to its* Exp *oracle. Then we can build adversary $\mathcal{A}_{\mathrm{KE}}$ such that*

$$\mathsf{Adv}^{\mathsf{ufexp}}_{\mathsf{DS}_{\mathrm{KU}}}(\mathcal{A}_{\mathsf{DS}}) \leq \mathsf{Adv}^{\mathsf{fsuf}}_{\mathsf{DS}_{\mathrm{KE}}}(\mathcal{A}_{\mathrm{KE}}). \tag{3}$$

Adversary $\mathcal{A}_{\mathrm{KE}}$ makes at most q_{UPD} queries to its Up *oracle, $q_{\mathrm{UPD}} + 1$ queries to its* Sign *oracle, and 1 query to its* Exp *oracle. The runtime of $\mathcal{A}_{\mathrm{KE}}$ is about that of $\mathcal{A}_{\mathsf{DS}}$.*

For simplicity, our theorems are given for single-user security (only one query to NewUser). A standard argument implies that our theorems extend to the multi-user security in a straightforward way with the same advantage bounds (using multi-user security of the underlying key-evolving digital signature scheme). Furthermore, since we consider only a single user, we can assume without loss of generality that adversary $\mathcal{A}_{\mathsf{DS}}$ in Theorem C.2 makes only one query to oracle Sign.

The formal proofs of the above theorems are straightforward, so we omit them. For Theorem C.1 note that a signature from $\mathsf{DS}_{\mathrm{KU}}$ essentially just consists of a sequence of signatures from $\mathsf{DS}_{\mathrm{KE}}$, specifically a signature $\mathbf{\Sigma}[i]$ for each Δ_i and a final σ_m for the actual message. If the two $\mathsf{DS}_{\mathrm{KU}}$ signatures are distinct either they differ in σ_m or in some $\mathbf{\Sigma}[i]$. Wherever they differ we immediately have two different $\mathsf{DS}_{\mathrm{KE}}$ signatures that verify for the same string. The proof is straightforward, so we omit it.

To prove Theorem C.2, note that adversary $\mathcal{A}_{\mathrm{KE}}$ (playing game FSUF) can perfectly simulate game UFEXP for adversary $\mathcal{A}_{\mathsf{DS}}$, answering the Upd, Sign, Exp oracle queries of the latter using the oracles Up, Sign, Exp of the former, respectively. As with our description above, whenever $\mathcal{A}_{\mathsf{DS}}$ successfully forges on $\mathsf{DS}_{\mathrm{KU}}$ we immediately get a forgery on $\mathsf{DS}_{\mathrm{KE}}$ by considering these sub-signatures.

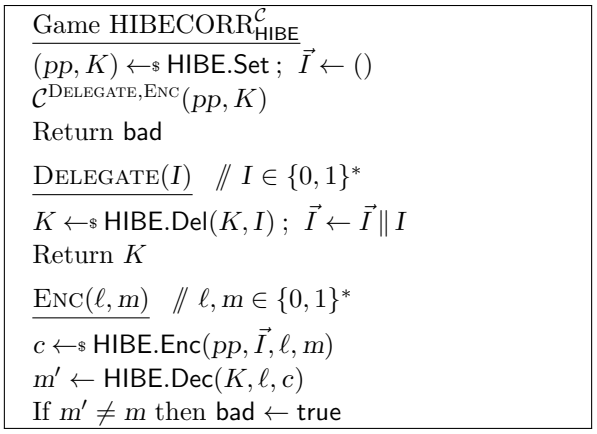

Figure 22: Games defining correctness of hierarchical public-key encryption scheme HIBE.

D Construction of key-updatable public-key encryption

In this section we more formally specify the construction of a key-updatable public-key encryption scheme that we sketched in Section 3.2.

Hierarchical identity based encryption schemes. In a hierarchical identity based encryption scheme, or HIBE [25], there are decryption keys (referred to as identity keys) associated with identities which are tuples of strings. Every user of the scheme can choose to encrypt to any identity knowing only that identity and some public parameters. Then anybody with an identity key for that identity can decrypt the ciphertext. Furthermore, the hierarchical part of HIBE means that any user with an identity key K for identity $\vec{I}$ can delegate a sub-key for any identity $\vec{I}'$ such that $\vec{I}$ is a prefix of $\vec{I}'$ (i.e. $\vec{I} \sqsubseteq \vec{I}'$).

More formally, HIBE scheme HIBE specifies algorithms HIBE.Set, HIBE.Del, HIBE.Enc, and HIBE.Dec. Setup algorithm HIBE.Set returns the public parameters pp and identity key K for the identity $\vec{I} = ()$. We write $(pp, K) \leftarrow\!\!\$\ \mathsf{HIBE.Set}$. Delegation algorithm HIBE.Del takes as input identity key K for the identity $\vec{I}$ and string $I \in \{0,1\}^*$ to return identity key K' for identity $\vec{I} \,\|\, I$. We write $K' \leftarrow\!\!\$\ \mathsf{HIBE.Del}(K, I)$. Encryption algorithm HIBE.Enc takes as input public parameters pp, identity $\vec{I}$, label ℓ, and message m to produce ciphertext c. We write $c \leftarrow\!\!\$\ \mathsf{HIBE.Enc}(pp, \vec{I}, \ell, m)$. Decryption algorithm HIBE.Dec takes as input identity key K, label ℓ, and ciphertext c to produce message $m \in \{0,1\}^*$. We write $m \leftarrow \mathsf{HIBE.Dec}(K, \ell, c)$. We let HIBE.EncRS denote the denote the set from which HIBE.Enc draws its random coins.

We denote the min-entropy of algorithms HIBE.Set and HIBE.Enc by $\text{H}_\infty(\mathsf{HIBE.Set})$ and $\text{H}_\infty(\mathsf{HIBE.Enc})$, respectively, defined as follows:

$$2^{-\text{H}_\infty(\mathsf{HIBE.Set})} = \max_{pp} \Pr\left[pp^* = pp : (pp^*, K^*) \leftarrow\!\!\$\ \mathsf{HIBE.Set}\right],$$

$$2^{-\text{H}_\infty(\mathsf{HIBE.Enc})} = \max_{pp,\vec{I},\ell,m,c} \Pr\left[c^* = c : c^* \leftarrow\!\!\$\ \mathsf{HIBE.Enc}(pp, \vec{I}, \ell, m)\right] .$$

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

Correctness. Correctness requires that if a message m is encrypted to identity $\vec{I}$ and K is an identity

Game $\text{HIBECCA}^{\mathcal{A}_{\mathsf{HIBE}}}_{\mathsf{HIBE}}$
$b \leftarrow\!\!\$\ \{0,1\}$; $S \leftarrow \emptyset$; $S_I \leftarrow \emptyset$
$(pp, K[0]) \leftarrow\!\!\$\ \mathsf{HIBE.Set}$; $\mathbf{idtable}[0] \leftarrow ()$
$b' \leftarrow\!\!\$\ \mathcal{A}^{\text{Enc,Dec,Delegate,Exp}}_{\mathsf{HIBE}}(pp)$
Return $(b = b')$

$\underline{\text{Enc}(\vec{I}, m_0, m_1, \ell)}$ // $\vec{I} \in (\{0,1\}^*)^*$, $m_0, m_1, \ell \in \{0,1\}^*$
If $|m_0| \neq |m_1|$ then return $\perp$
If $\exists \vec{I}' \in S_I$ s.t. $\vec{I}' \sqsubseteq \vec{I}$ then return $\perp$
$c \leftarrow\!\!\$\ \mathsf{HIBE.Enc}(pp, \vec{I}, \ell, m_b)$
$S \leftarrow S \cup \{(\vec{I}, c, \ell)\}$
Return c

$\underline{\text{Dec}(\Upsilon, c, \ell)}$ // $\Upsilon, c, \ell \in \{0,1\}^*$
If $K[\Upsilon] = \perp$ then return $\perp$
$v \leftarrow (\mathbf{idtable}[\Upsilon], c, \ell)$
If $\exists (\vec{I}', c', \ell') \in S$ s.t. $(\vec{I}', c', \ell') = v$ then return $\perp$
$m \leftarrow \mathsf{HIBE.Dec}(K[\Upsilon], \ell, c)$
Return m

$\underline{\text{Delegate}(\Upsilon, \Upsilon', I)}$ // $\Upsilon, \Upsilon', I \in \{0,1\}^*$
If $K[\Upsilon] = \perp$ then return $\perp$
If $K[\Upsilon'] \neq \perp$ then return $\perp$
$K[\Upsilon'] \leftarrow\!\!\$\ \mathsf{HIBE.Del}(K[\Upsilon], I)$
$\mathbf{idtable}[\Upsilon'] \leftarrow \mathbf{idtable}[\Upsilon] \,\|\, I$
Return $\perp$

$\underline{\text{Exp}(\Upsilon)}$ // $\Upsilon \in \{0,1\}^*$
If $K[\Upsilon] = \perp$ then return $\perp$
If $\exists (\vec{I}, c, \ell) \in S$ s.t. $\mathbf{idtable}[\Upsilon] \sqsubseteq \vec{I}$ then
  Return $\perp$
$S_I \leftarrow S_I \cup \{\mathbf{idtable}[\Upsilon]\}$
Return $K[\Upsilon]$

Figure 23: Games defining CCA security of hierarchical public-key encryption scheme HIBE.

Algorithm $\mathsf{PKE.Kg}$
$(pp, K) \leftarrow\!\!\$\ \mathsf{HIBE.Set}$; $ek \leftarrow (pp, ())$
Return (ek, K)

Algorithm $\mathsf{PKE.Enc}(ek, \ell, m)$
$(pp, \vec{I}) \leftarrow ek$; $c \leftarrow\!\!\$\ \mathsf{HIBE.Enc}(pp, \vec{I}, \ell, m)$
Return c

Algorithm $\mathsf{PKE.UpdEk}(ek, \Delta)$
$(pp, \vec{I}) \leftarrow ek$; $ek \leftarrow (pp, \vec{I} \,\|\, \Delta)$
Return ek

Algorithm $\mathsf{PKE.Dec}(dk, \ell, c)$
$m \leftarrow \mathsf{HIBE.Dec}(dk, \ell, c)$
Return (dk, m)

Algorithm $\mathsf{PKE.UpdDk}(dk, \Delta)$
$dk \leftarrow\!\!\$\ \mathsf{HIBE.Del}(dk, \Delta)$
Return dk

Figure 24: Key-updatable public-key encryption scheme $\mathsf{PKE} = \mathsf{PKE\text{-}CONS}[\mathsf{HIBE}]$.

key for that identity then K will decrypt the ciphertext properly. This is formalized by the game HIBECORR shown in Fig. 22. In it the adversary is given the public parameters pp and identity key K. It can make calls to Delegate with a string I to ask for K to be updated to the identity $\vec{I} = \vec{I} \,\|\, I$. It will be given the new K. Finally it can query a label ℓ and message m to Enc. The message and label are then encrypted to the current identity $\vec{I}$. The produced ciphertext is immediately decrypted by the corresponding identity key K. If the decryption does not return the message that was encrypted then bad is set true. The adversary wins if it can cause bad to be set true.

The advantage of adversary $\mathcal{C}$ is defined by $\mathsf{Adv}^{\mathsf{hibecorr}}_{\mathsf{HIBE}}(\mathcal{C}) = \Pr[\text{HIBECORR}^{\mathcal{C}}_{\mathsf{HIBE}}]$. Perfect correctness requires that $\mathsf{Adv}^{\mathsf{hibecorr}}_{\mathsf{HIBE}}(\mathcal{C}) = 0$ for all (even unbounded) adversaries.

IND-CCA security of HIBE. For security we will require that CCA security hold even when the adversary is given identity keys for arbitrarily many identities, as long as none of these identities are prefixes of any identities it made a challenge query to. Security is formally defined by the game

HIBECCA shown in Fig. 23.

In this game an adversary $\mathcal{A}_{\mathsf{HIBE}}$ is given the public parameters pp and then has access to five oracles. The initial identity key with identity () is stored in $K[0]$. The adversary can ask for a sub-key to be delegated from an existing identity key $K[\Upsilon]$ with identity $\vec{I}$ by calling DELEGATE with Υ, an unused key identifier Υ', and a string I. This oracle will create a new identity key $K[\Upsilon']$ with identity $\vec{I} \,\|\, I$. The adversary can ask for the encryption of a challenge message using ENC to any identity $\vec{I}$ by calling it with input $\vec{I}$, m_0, m_1, and ℓ as long as no identity keys have been exposed for an identity that is a prefix of $\vec{I}$. The adversary can ask DEC for the decryption of any (c, ℓ) pair by any existing identity key $K[\Upsilon]$ as long as the pair was not the output of a prior encryption query to the identity of $K[\Upsilon]$. Finally, the adversary can use EXP to ask for the value of any identity key $K[\Upsilon]$ (which is associated with some identity $\vec{I}$) as long as it has not asked a challenge encryption query to an identity which $\vec{I}$ is a prefix of. The goal of the adversary is to guess the secret bit corresponding to which message ENC encrypts.

Adversary $\mathcal{A}_{\mathsf{HIBE}}^{\text{ENC,DEC,DELEGATE,EXP}}(pp)$	$\underline{\text{ENCS}(\Lambda, m_0, m_1, \ell)}$
$b' \leftarrow\!\!\$\ \mathcal{B}_{\mathsf{PKE}}^{\text{NEWUSERS,UPDEKS,UPDDKS,ENCS,DECS,EXPS}}$	If $\lvert m_0\rvert \neq \lvert m_1\rvert$ then return $\perp$
Return b'	If $\vec{\Delta}'[\Lambda] \sqsubseteq \vec{\Delta}_e[\Lambda]$ then return $\perp$
$\underline{\text{NEWUSERS}(\Lambda)}$	$c \leftarrow \text{ENC}(\vec{\Delta}_e[\Lambda], m_0, m_1, \ell)$
$\vec{\Delta}_e[\Lambda] \leftarrow () \,;\ \vec{\Delta}_d[\Lambda] \leftarrow ()$	$S[\Lambda] \leftarrow S[\Lambda] \cup \{(\vec{\Delta}_e[\Lambda], c, \ell)\}$
$\vec{\Delta}'[\Lambda] \leftarrow \perp \,;\ S[\Lambda] \leftarrow \emptyset$	Return c
$ek[\Lambda] \leftarrow (pp, \vec{\Delta}_e[\Lambda])$	$\underline{\text{DECS}(\Lambda, c, \ell)}$
Return $ek[\Lambda]$	If $(\vec{\Delta}_d[\Lambda], c, \ell) \in S[\Lambda]$ then return $\perp$
$\underline{\text{UPDEKS}(\Lambda, \Delta)}$	$m \leftarrow \text{DEC}(\lvert\vec{\Delta}_d[\Lambda]\rvert, c, \ell)$
$\vec{\Delta}_e[\Lambda] \leftarrow \vec{\Delta}_e[\Lambda] \,\|\, \Delta$	Return m
$\underline{\text{UPDDKS}(\Lambda, \Delta)}$	$\underline{\text{EXPS}(\Lambda)}$
$l \leftarrow \lvert\vec{\Delta}_d[\Lambda]\rvert \,;\ \text{DELEGATE}(l, l+1, \Delta)$	If $\exists(\vec{\Delta}, c, \ell) \in S[\Lambda]$ s.t. $\vec{\Delta}_d[\Lambda] \sqsubseteq \vec{\Delta}$ then return $\perp$
$\vec{\Delta}_d[\Lambda] \leftarrow \vec{\Delta}_d[\Lambda] \,\|\, \Delta$	If $\vec{\Delta}'[\Lambda] = \perp$ then $\vec{\Delta}'[\Lambda] \leftarrow \vec{\Delta}_d[\Lambda]$
	Return $\text{EXP}(\lvert\vec{\Delta}_d[\Lambda]\rvert)$

Figure 25: Adversary $\mathcal{A}_{\mathsf{HIBE}}$ used for proof of Theorem D.1.

To keep track of the various things $\mathcal{A}_{\mathsf{HIBE}}$ is not allowed to do the game keeps a table **idtable** which maps key identifiers to the identity of that key, set S which stores all of the tuples $(\vec{I}, c, \ell)$ where (c, ℓ) was the output of an encryption query to identity $\vec{I}$, and set S_I which stores the identities of all keys that have been exposed. The advantage of adversary $\mathcal{A}_{\mathsf{HIBE}}$ is defined by $\mathsf{Adv}^{\mathsf{hibecca}}_{\mathsf{HIBE}}(\mathcal{A}_{\mathsf{HIBE}}) = 2\Pr[\text{HIBECCA}^{\mathcal{A}_{\mathsf{HIBE}}}_{\mathsf{HIBE}}] - 1$.

Key-updatable public-key encryption scheme PKE-CONS[HIBE]. Let HIBE be a hierarchical identity based encryption scheme. We build a key-updatable public-key encryption scheme PKE-CONS[HIBE] as defined in Fig. 24. It essentially corresponds to using the HIBE directly by setting $ek = (pp, \vec{I})$ and $dk = K$. It is clear that $\mathrm{H}_\infty(\mathsf{PKE.Kg}) = \mathrm{H}_\infty(\mathsf{HIBE.Set})$ and $\mathrm{H}_\infty(\mathsf{PKE.Enc}) = \mathrm{H}_\infty(\mathsf{HIBE.Enc})$.

The following theorem bounds the advantage of an adversary attacking PKE-CONS[HIBE] by the advantage of a similarly efficient adversary in attacking the security of HIBE. The theorem is relatively straightforward because the security of a key-updatable public key encryption scheme is essentially a special case of HIBE security.

Theorem D.1 *Let* HIBE *be a hierarchical identity based encryption scheme and let* PKE *denote* PKE-CONS[HIBE]. *Let* $\mathcal{A}_{\mathsf{PKE}}$ *be an adversary making at most* 1 *query to its* NewUser *oracle,* q_{UpdEk} *queries to its* UpdEk *oracle,* q_{UpdDk} *queries to its* UpdDk *oracle,* q_{Enc} *queries to its* Enc *oracle,* q_{Dec} *queries to its* Dec *oracle, and* q_{Exp} *queries to its* Exp *oracle. Then we can build adversary* $\mathcal{A}_{\mathsf{HIBE}}$ *against the security of* HIBE *such that*

$$\mathsf{Adv}^{\mathsf{indexp}}_{\mathsf{PKE}}(\mathcal{A}_{\mathsf{PKE}}) \leq \mathsf{Adv}^{\mathsf{hibecca}}_{\mathsf{HIBE}}(\mathcal{A}_{\mathsf{HIBE}}). \tag{4}$$

Adversary $\mathcal{A}_{\mathsf{HIBE}}$ *makes at most* q_{Enc} *queries to its* Enc *oracle,* q_{Dec} *queries to its* Dec *oracle,* q_{UpdDk} *queries to its* Delegate *oracle, and* q_{Exp} *queries to its* Exp *oracle. The running time of* $\mathcal{A}_{\mathsf{HIBE}}$ *is about that of* $\mathcal{A}_{\mathsf{PKE}}$.

For simplicity we show that single-user security of PKE (i.e. $\mathcal{A}_{\mathsf{PKE}}$ makes at most 1 query to is NewUser oracle) is obtained from the single-user security of HIBE. This tight reduction between two single-user security notions can be generically transformed into a tight reduction between the corresponding multi-user notions.

Proof of Theorem D.1: Because $\mathcal{A}_{\mathsf{PKE}}$ makes at most one query to NewUser, we will without loss of generality consider the adversary an $\mathcal{B}_{\mathsf{PKE}}$ which makes its first query to NewUser and makes all queries with a single, fixed value of Λ because there must exist such an adversary with efficiency about that of $\mathcal{A}_{\mathsf{PKE}}$ which satisfies $\mathsf{Adv}^{\mathsf{indexp}}_{\mathsf{PKE}}(\mathcal{A}_{\mathsf{PKE}}) \leq \mathsf{Adv}^{\mathsf{indexp}}_{\mathsf{PKE}}(\mathcal{B}_{\mathsf{PKE}})$.

Now consider the adversary $\mathcal{A}_{\mathsf{HIBE}}$ shown in Fig. 25. It simulates the view of $\mathcal{A}_{\mathsf{PKE}}$ in the obvious way using its HIBECCA oracles. Its code was obtained by plugging the code of PKE into INDEXP and then, where appropriate, replacing executing algorithms of HIBE with oracle queries by $\mathcal{A}_{\mathsf{HIBE}}$ and removing code which is irrelevant for $\mathcal{B}_{\mathsf{PKE}}$.

The underlying secret bit of HIBECCA plays the role of the secret bit in INDEXP. We need to argue that the simulated view of $\mathcal{A}_{\mathsf{PKE}}$ is identical to its view in INDEXP. This would be immediate to verify if the various oracles in HIBECCA always returned their intended value; however, sometimes the abort early, returning $\perp$. We will analyze the possible ways of this occurring individually.

For all of the oracles in HIBECCA other than have checks about whether entries of $K[\Upsilon]$ have or have not been initialized yet. Note that $\mathcal{A}_{\mathsf{HIBE}}$ uses the lengths of $\vec{\Delta}_d[\Lambda]$. This is incremented by 1 with each UpdDkS query. Based on this we can verify that these checks will never cause the oracles of HIBECCA to abort early when called by $\mathcal{A}_{\mathsf{HIBE}}$.

Adversary $\mathcal{A}_{\mathsf{HIBE}}$ will never query Enc with m_0 and m_1 of different length, so the corresponding check in Enc will never cause it to abort early.

The remaining checks we must analyze are those that depend on the sets S and S_I. Note that for any $\vec{I} \in S_I$ it will hold that $\vec{\Delta}'[\Lambda] \sqsubseteq \vec{I}$. Similarly for any $(\vec{I}, c, \ell) \in S$ it will hold that $(\vec{I}, c, \ell)S[\Lambda]$. From this it is clear that the checks $\mathcal{A}_{\mathsf{HIBE}}$ performs before making oracle queries will prevent any of its oracles from returning $\perp$ because of the checks depending on these sets.

Hence it is clear that $\mathcal{A}_{\mathsf{HIBE}}$ correctly guesses the secret bit whenever $\mathcal{B}_{\mathsf{PKE}}$ would guess its secret bit, so $\mathsf{Adv}^{\mathsf{indexp}}_{\mathsf{PKE}}(\mathcal{B}_{\mathsf{PKE}}) \leq \mathsf{Adv}^{\mathsf{hibecca}}_{\mathsf{HIBE}}(\mathcal{A}_{\mathsf{HIBE}})$, completing the proof. ∎