A Unified and Composable Take on Ratcheting

Department of Computer Science, ETH Zurich, 8092 Zurich, Switzerland.
{dajost, maurer, mumarta}@inf.ethz.ch

Abstract. Ratcheting, an umbrella term for certain techniques for achieving secure messaging with strong guarantees, has spurred much interest in the cryptographic community, with several novel protocols proposed as of lately. Most of them are composed from several sub-protocols, often sharing similar ideas across different protocols. Thus, one could hope to reuse the sub-protocols to build new protocols achieving different security, efficiency, and usability trade-offs. This is especially desirable in view of the community's current aim for group messaging, which has a significantly larger design space. However, the underlying ideas are usually not made explicit, but rather implicitly encoded in a (fairly complex) security game, primarily targeted at the overall security proof. This not only hinders modular protocol design, but also makes the suitability of a protocol for a particular application difficult to assess.

In this work we demonstrate that ratcheting components can be modeled in a composable framework, allowing for their reuse in a modular fashion. To this end, we first propose an extension of the Constructive Cryptography framework by so-called global event histories, to allow for a clean modularization even if the component modules are not fully independent but actually subtly intertwined, as in most ratcheting protocols. Second, we model a unified, flexibly instantiable type of strong security statement for secure messaging within that framework. Third, we show that one can phrase strong guarantees for a number of sub-protocols from the existing literature in this model with only minor modifications, slightly stronger assumptions, and reasonably intuitive formalizations.

When expressing existing protocols' guarantees in a simulation-based framework, one has to address the so-called commitment problem. We do so by reflecting the removal of access to certain oracles under specific conditions, appearing in game-based security definitions, in the real world of our composable statements. We also propose a novel non-committing protocol for settings where the number of messages a party can send before receiving a reply is bounded.

1 Introduction

1.1 Secure Messaging and Ratcheting

Secure-messaging (SM) protocols attempt to provide strong security guarantees to two parties that communicate over an asynchronous network. Apart from

* Research supported by the Zurich Information Security and Privacy Center (ZISC).

protecting confidentiality and integrity of messages, the desired properties include forward secrecy and healing from a state or randomness exposure. The latter properties are addressed by the so-called ratcheting protocols, by having the parties continuously update their secret keys.

The term ratcheting on its own does not carry any formal meaning; rather, it is an umbrella term for a number of different guarantees, somehow related to the concept of updating keys. One notable example of ratcheting is the widely-used Signal protocol [20] with its double-ratchet algorithm, formally analyzed in [7, 1]. Furthermore, there exist protocols with much stronger guarantees, but that require the messages to be delivered in order [23, 9, 8, 10]. Protocols with the stronger guarantee of immediate out-of-order decryption have been proposed in [1]. While the majority of the literature considers secure communication, some works view ratcheting as a property of key exchange instead [2, 23].

A number of proposed protocols pursue similar goals, but each achieves a slightly different trade-off between security, efficiency and usability. Moreover, each construction comes with its own—usually fairly complex—security game, intermediate abstractions, and primitives. This renders them hard to compare and hinders achieving new trade-offs that would result from combining ideas from different protocols. This motivates the goal of this work, which is to facilitate a systematic, modular and composable analysis of secure-messaging protocols.

1.2 Composable Security

While a game-based systematization of secure messaging could certainly address some of the aforementioned concerns, composable frameworks, such as [4, 22, 17, 13], provide some distinct advantages.

First, security under (universal) composition is a stronger notion: the guarantees are provided even if a protocol is executed in an arbitrary environment, alongside other protocols. So far, no SM protocol provably achieves this (in fact, even the weaker notion of security under parallel self composition has not been analyzed). Moreover, composable frameworks facilitate modularity. One can define components with clean abstraction boundaries (e.g., a secure channel) and use their idealized versions in a higher-level protocol (and its proof). The overall security of the composed protocol follows from the composition theorem. This stands in contrast with game-based definitions, where security of the components and the overall protocol is expressed by a number of games, and one has to show that winning the security game for the overall protocol implies, via reductions, winning one security game for a component. Finally, guarantees expressed in a composable framework usually have more evident semantics, obtained from directly considering how a protocol is used, rather than a hypothetical interaction of an adversary with a simplified game that encodes excluded attacks.

Unfortunately, secure messaging does not render itself easily to a modular, composable analysis. One reason for this is the difficulty in drawing the right abstraction boundaries. Roughly, the guarantees for a channel heavily depend on other components in the system, for example, we may want to say that the confidentiality of a message is protected only if some memory contents do not

leak. This problem also appears in the analysis of some protocols from different contexts (e.g. TLS [11]), which often violate the rules of modularity.

Furthermore, we encounter the so-called "commitment problem" of simulation-based security definitions. Intuitively, the natural composable guarantees are too strong and provide additional security that seems to carry little significance in practice, and that can only be achieved with (stronger) setup assumptions and at an efficiency loss. To address this problem, a number of approaches have been proposed — none of them, however, being able to fully satisfactorily formalize the weaker guarantees achieved by regular schemes. First, the notion of non-information oracles [6] has been proposed that essentially embeds a game-base definition in a composable abstraction module. Second, a line of work considers stronger, i.e., super-polynomial, simulators [21, 24, 3]. Protocols in those models, however, still have to rely on additional setup and special primitives.

1.3 Contributions

This paper makes both conceptual and technical contributions. Conceptual contributions to composable security frameworks are the notion of global event histories as well as a modeling technique for circumventing the so-called commitment problem. Technical contributions include the modeling of ratcheting (sub-)protocols in a composable framework as well as a novel protocol that achieves adaptive security, i.e., the strongest form of composable security, under certain restrictions.

Global event histories. Composable frameworks are based around the idea of independent modules (e.g. channels, keys, or memory resources) that are constructed by one protocol and then used by another protocol in the next construction step. However, in many settings, in particular as they occur in modeling ratcheting protocols, these components are subtly correlated which seems to violate modularity. For example, a channel (one module) can become insecure when a key (another, apparently independent module) is leaked to the adversary.

We address this problem by two conceptual ideas. First, we parameterize resources by several (discrete) parameters—which can be thought of as a switch with two or more positions—which can downgrade the security of a resource, e.g. switch a channel from non-leakable (i.e. confidential) to leakable. Second, we introduce the notion of *global event histories* defined for the entire real (or ideal) world, where a history is a list of events having happened at a module (e.g. a message being input by Alice or a message having leaked to the adversary). A key idea is now that the switch settings of the modules can be defined by predicates (or, more generally, multi-valued functions) of the global event history. This allows us to draw meaningful abstraction boundaries for secure messaging, but we believe that the concept of event histories is of independent interest and may enable modular analyses for settings where this was previously difficult.

Formally, we use the Constructive Cryptography (CC) framework [17, 14], and in particular a slight modification of its standard instantiation to model the event history. Since the composition theorem of CC is proved on an abstract level, we do not need to re-prove it.

Expressing the guarantees provided by ratcheting. Our goal is to capture the guarantees provided by ratcheting (sub-)protocols in a general fashion to make them reusable in different protocols or contexts. This is in contrast to existing game-based definitions, which usually formalize exactly what is required by the next sub-protocol for the overall protocol's security proof to go through.

This goal is achieved by considering parameterized resources as described above and modeling the goal of a (sub-)protocol as improving a certain parameter while leaving the other parameters unchanged, independently of what they are. One can think of a protocol improving certain switch positions (e.g. making a channel confidential), independently of the other switch positions.

In this paper, we consider three ratcheting sub-protocols. We start with a simple authentication protocol in the unidirectional setting which constantly updates keys. As a more involved example, we consider the use of hierarchical identity-based encryption to provide confidentiality. As a third example, we analyze continuous key agreement, a notion introduced by Alwen et al. [1] to abstract the symmetric ratcheting layer of Signal. On the way, we discover cases where the existing game-based notions are insufficient to prove the stronger, more modular, statements that don't fix the properties (i.e., the switch positions) of the assumed network, but where they can be achieved by simple modifications.

Solutions to the commitment problem. When modeling ratcheting protocols, we encounter the so-called commitment problem: the simulator would have to output a simulated value (e.g. a ciphertext) which at a later stage must be compatible with another value (e.g. a leaked key) not initially known to him. Since this is generally impossible, we address this problem in two alternative ways.

On one hand, we propose a technique that allows to transform many standard SM protocols into protocols that achieve full composable security, at the expense of an efficiency lost, as well as being restricted to only sending a bounded number of messages before receiving a reply from the other party. We apply this technique to the HIBE protocol mentioned above and construct its fully composable version.

On the other hand, we can retain composable statements of regular protocols by restricting the adversary's capabilities in the real world as a function of the event history. Roughly, some real-world components become secure by assumption after certain sequences of events. This is similar to how some oracles (e.g., those that expose a secret key) are disabled in game-based definitions. We hence give such special conditions in games a composable semantics.

1.4 Outline

In Section 3 we extend Constructive Cryptography to include the global event history. Based on this, in Section 4, we introduce a simple and generic type of security statement for SM protocols. (In Section 7 we extend it to encompass ratcheting as a key-exchange primitive.) In Section 5, we demonstrate how the security guarantees of ratcheting components can be phrased in this model. Finally, in Section 6, we introduce a novel non-committing ratcheting protocol that achieves full simulation-based security for a bounded number of messages.

2 Preliminaries: Constructive Cryptography

2.1 The Real-World / Ideal-World Paradigm

Many security definitions, and in particular most composable security frameworks [4, 22, 17, 13], are based on the real-world/ideal-world paradigm. The real world models the use of a protocol, whereas the ideal world formalizes the security guarantees that this protocol is supposed to achieve.

The security statement then affirms that the real word is "just-as-good" as the ideal world, meaning that for all parties, no matter whether honest or adversarial, it does not make a difference whether they live in the real or ideal world. Hence, if the honest parties are content with the guarantees they get in the ideal world, they can safely execute the protocol in the real world instead.

2.2 Resources

In each composable framework there is some notion of a module that exports a well-defined interface in a black-box manner to the rest of the world. In the UC framework such a module is called a *functionality*. In the Constructive Cryptography (CC) framework [17, 14] such a module is called a *resource*. One of the main differences is that in CC a world consists entirely of resources and the environment (called a distinguisher). So while UC distinguishes between the real world, where the parties can only send messages to each other, and a hybrid world, where they additionally access some ideal functionalities, in CC everything, including communication, is a resource. For example, a security statement about two parties using authenticated encryption to transmit a message is phrased as a real world containing two resources—an insecure channel and a shared key—which are then used by the protocol to construct the ideal world consisting of a secure channel. See Figure 1 for a description of the real-world resources.

A resource is a reactive system that allows interaction at one or several *interfaces*, i.e. upon providing an input at one of the interfaces, the system provides an output. In this work, we only consider systems where the output is produced at the same interface the input was given. Formally, resources are modeled as random systems [15], where the interface address is encoded as part of the inputs. However, a reader unfamiliar with CC may simply think of a resource with n interfaces as n oracles that share a joint state. Note that there is no formal notion of a party in constructive cryptography; they only give meaning to the construction statements, by thinking of each interface being controlled by some party. Since in this work we make statements about messaging between two honest parties, called Alice and Bob, in the presence of a global adversary, called Eve, we usually label the interfaces accordingly, indicating how the assignment of interfaces to parties should be understood.

A set of resources can be composed into a single one. The interface set of the composed resource corresponds to the union of the ones from the composed resources. Returning to our example of authenticated encryption, in the real world we have both an insecure channel $\mathsf{InsecCh}$ and a key Key, where the former has

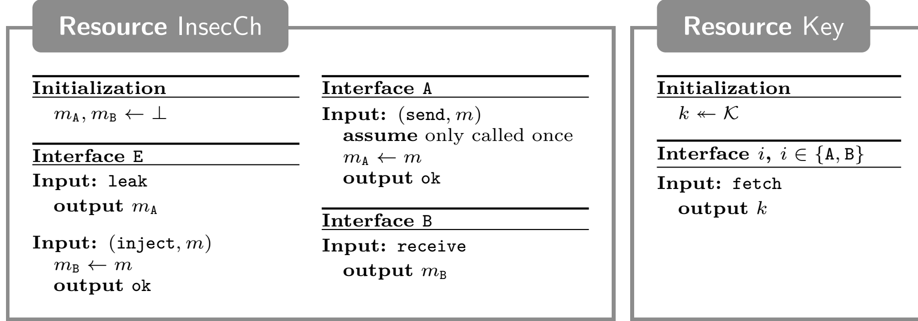

Fig. 1: The assumed real-world resources of the authenticated-encryption example: an insecure channel and a shared key. The insecure channel exports three interface A, B, and E, understood to be controlled by the respective parties Alice, Bob, and Eve, whereas the key resource only exports two interfaces.

three interfaces and the latter two. The composed resource, denoted $[\mathsf{InsecCh}, \mathsf{Key}]$, is a resource with five interfaces, each of them addressed by a tuple consisting of the resource's name and the interface's original name.

We describe our resources using pseudo-code (c.f. Figure 1). The following conventions are followed: each resource has an initialization procedure initializing all the persistent variables (all other variables are understood to be volatile). Formally this initialization is called upon invoking any arbitrary interface for the first time. Each interface exposes one or more capabilities, each of them described by a keyword (e.g. `send` in case of a channel), and the (potential empty) list of arguments (e.g., m). Furthermore, we use the **assume** command, which should be understood as a shortcut for explicitly tracking the respective condition and returning an error symbol $\bot$ in case the condition is violated. In Figure 1, the keyword **assume** is used the specify that the channel is single-use.

2.3 Converters

The protocol execution in CC is modeled by *converters*, each of which expresses the local computation executed by one party. (The name converter derives from the property that a converter attached to a resource converts it into another "ideal" one.) A converter expects to be connected to a given set of interfaces at the "inside", and emulates a certain set of interfaces at the "outside". Upon an input at one of the emulated interfaces, the converter is allowed to make a bounded number of oracle queries to the inside interfaces (recall that a resource always returns at the same interface it was queried), before returning a value at the same emulated interface. For a converter prot and a resource R, we denote by $\mathsf{R}' := \mathsf{prot}^{\{I_1,\dots,I_n\}}\mathsf{R}$ the resource obtained from connecting the converter to the subset $\{I_1, \dots, I_n\}$ of the interfaces. The resource R' no longer exposes those interfaces to the world, but the ones emulated by prot instead. We usually omit

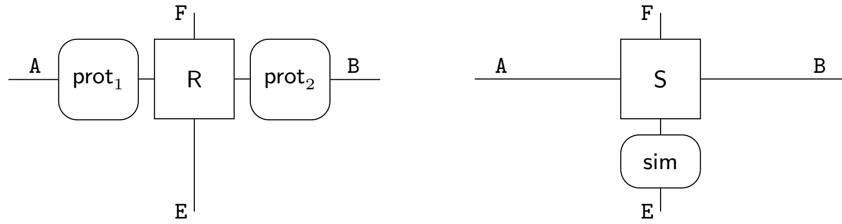

Fig. 2: Execution of the protocol in the real world by Alice and Bob (left) and the ideal world with the simulator attached to Eve's interface (right). The "free" interface on the top is accessed directly by the environment in both worlds.

specifying the set $\{I_1, \ldots, I_n\}$ and just write for instance $\mathsf{prot}^{\mathtt{A}}\mathsf{R}$, denoting that it is connected to all of Alice's interfaces.

2.4 The Construction Notion

Security is then defined following the real-world/ideal-world paradigm, stating that in every environment the real world should behave the same way as the ideal one. The real world, as depicted in Figure 2, thereby consists of the assumed resource R to which the converters are attached, each to a subset of the respective party's interfaces. The ideal world, on the other hand, consists of the constructed resource S with a simulator (which is a converter) attached to Eve's interfaces.

Behaving the same way is formalized using the notion of a distinguisher, that can make oracle queries to the resource's interfaces and then outputs a bit, indicating whether it believes to interact with the real or ideal world. More formally, in the special case of two honest parties Alice and Bob and a global adversary Eve, the goal of a distinguisher $\mathbf{D}$ is to distinguish the real world $\mathsf{prot}_1^{\mathtt{A}}\, \mathsf{prot}_2^{\mathtt{B}}\, \mathsf{R}$ from the ideal world $\mathsf{sim}^{\mathtt{E}}\, \mathsf{S}$. The advantage of $\mathbf{D}$ is defined as

$$\varDelta^{\mathbf{D}}\big(\mathsf{prot}_1^{\mathtt{A}}\, \mathsf{prot}_2^{\mathtt{B}}\, \mathsf{R},\ \mathsf{sim}^{\mathtt{E}}\, \mathsf{S}\big) := \Pr\big[\mathbf{D}(\mathsf{sim}^{\mathtt{E}}\, \mathsf{S}) = 1\big] - \Pr\big[\mathbf{D}(\mathsf{prot}_1^{\mathtt{A}}\, \mathsf{prot}_2^{\mathtt{B}}\, \mathsf{R}) = 1\big].$$

Let ϵ denote a function mapping distinguishers to values in $[-1, 1]$. Then, the protocol $(\mathsf{prot}_1, \mathsf{prot}_2)$, when attached to $\mathtt{A}$ and $\mathtt{B}$, is said to *construct* S from R within ϵ, and with respect to sim attached to $\mathtt{E}$, if

$$\forall \mathbf{D} : \varDelta^{\mathbf{D}}\big(\mathsf{prot}_1^{\mathtt{A}}\, \mathsf{prot}_2^{\mathtt{B}}\, \mathsf{R},\ \mathsf{sim}^{\mathtt{E}}\, \mathsf{S}\big) \leq \epsilon(\mathbf{D}).$$

Note that we require the sets of interfaces controlled by Alice, Bob, and Eve, respectively, to be pairwise disjoint. They however do not have to completely partition the set of interfaces. The remaining interfaces are called *free interfaces* to which the distinguisher has direct access in both worlds.

For simplicity, in this work we consider an asymptotic setting only (although we usually do not make the asymptotics explicit) where all resources and converters are assumed to be efficiently implementable. We then write

$$\mathsf{prot}_1^{\mathtt{A}}\, \mathsf{prot}_2^{\mathtt{B}}\, \mathsf{R} \approx \mathsf{sim}^{\mathtt{E}}\, \mathsf{S},$$

if $\Delta^{\mathbf{D}}\big(\mathsf{prot}_1^{\mathsf{A}}\,\mathsf{prot}_2^{\mathsf{B}}\,\mathsf{R},\ \mathsf{sim}^{\mathsf{E}}\,\mathsf{S}\big)$ is negligible for every efficient distinguisher $\mathbf{D}$, and simply say that $(\mathsf{prot}_1, \mathsf{prot}_2)$ constructs S from R if there exists an efficient simulator sim achieving this.

Note that the notion of construction is analogous to the notion of secure realization in the UC framework. In contrast to UC, however, the set of all resource instances within a construction statement is fixed. The distinguisher does not instantiate resources or protocols, or assign session identifiers. Dynamic availability properties of resources can obviously still be modeled as part of the resources themselves, though.

2.5 Composition

The notion of construction is composable, which intuitively means that if a protocol $(\mathsf{prot}_1, \mathsf{prot}_2)$ constructs S from R, and another protocol $(\mathsf{prot}'_1, \mathsf{prot}'_2)$ constructs T from S, then the combined protocol constructs T from R. This is known as *sequential composition*. Additionally, if $(\mathsf{prot}_1, \mathsf{prot}_2)$ constructs $[\mathsf{S}_1, \dots, \mathsf{S}_i]$ from $[\mathsf{R}_1, \dots, \mathsf{R}_j]$, for some i and j, then for every set of (efficiently implementable) resources $\{\mathsf{T}_1, \dots, \mathsf{T}_n\}$ it also holds that $(\mathsf{prot}_1, \mathsf{prot}_2)$ constructs $[\mathsf{S}_1, \dots, \mathsf{S}_i, \mathsf{T}_1, \dots, \mathsf{T}_n]$ from $[\mathsf{R}_1, \dots, \mathsf{R}_j, \mathsf{T}_1, \dots, \mathsf{T}_n]$, where the interfaces of the additional resources $\mathsf{T}_1, \dots, \mathsf{T}_n$ are treated as free in the construction. This property is known as *parallel composition*.

Both properties are proven in [17, 18] for a more abstract notion of resources being "just-as-good", of which the here introduced indistinguishability notion is a special case. Together, the two properties form the equivalent to the universal composability property of the UC framework.

3 Constructive Cryptography with Events

In this section we generalize the Constructive Cryptography framework to allow for better modularization. More specifically, we introduce another instantiation of resources and the "just-as-good" notion, thereby inheriting the composition theorem of CC that is proven on a more abstract level.

Motivation. Recall that SM protocols are difficult to modularize, because the guarantees for a given message depend on the dynamically changing state of other components in the system, such as whether the state leaked or the adversary tampered with a previous message. In traditional CC, where the abstraction boundary of a resource is just the input-output behavior, properly accounting for those dependencies would essentially force us to model the whole SM application as monolithic resource. In this section, we therefore extend the notions of resources and construction to relax the abstraction boundary in a clean and well-controlled manner, which will allow for such dependencies between different resources. More concretely, we introduce a global event history. Each resource is then allowed to trigger events from a predefined set (e.g. indicating that a party's state leaked),

on which the behavior of other resources can then depend. The event history is visible to the environment, the resources, and the simulator.[1]

The global event history. We model events as a generalization of monotone binary outputs (MBO) introduced by Maurer et al. [16]. Roughly, an MBO of a resource is an additional output that can change from 0 to 1 but not back. This can be interpreted as a single event, which happens when the MBO changes to 1. We generalize this to many events by the means of a global event history.

Definition 1. *Let $\mathcal{N}$ be a name set. The* global event history *$\mathcal{E}$ is a list of elements of $\mathcal{N}$ without duplicates.*

For $n \in \mathcal{N}$, we use $\mathcal{E}_n$ as a short-hand notation to denote that n is in the list $\mathcal{E}$, and say that the event happened. Analogously, we use $\neg\mathcal{E}_n$ to denote the complementary case. Furthermore, we denote by $\mathcal{E} \xleftarrow{+} \mathcal{E}_n$, the act of appending n to the list $\mathcal{E}$, if $\neg\mathcal{E}_n$, and leaving the list unchanged otherwise.

We also introduce the natural happened-before relation on the events.

Definition 2. *For $n_1, n_2 \in \mathcal{N}$, we say that the event n_1 precedes the event n_2 in the event history $\mathcal{E}$, denoted $\mathcal{E}_{n_1} \prec \mathcal{E}_{n_2}$, if either*

- *both events happened, i.e, $\mathcal{E}_{n_1}$ and $\mathcal{E}_{n_2}$, and n_1 is in the history before n_2,*
- *or only n_1 happened so far.*

Note that saying that $\mathcal{E}_{n_1} \prec \mathcal{E}_{n_2}$ is true if so far only the former one has happened best matches the type of statement we usually want to make: for instance, if we express the condition that a message is secure if the key has been securely erased before the memory was leaked, then we do not need to insist that the memory actually leaked.

Event-aware systems. We consider resources, converters and distinguishers that can (1) read the global event history, and (2) append to the event history from a fixed subset of $\mathcal{N}$. That is, the global event history is an additional component (of both the real and ideal world) that models event-awareness in an abstract manner, rather than formalizing them as outputs that need to be explicitly passed between components.

As a convention, we use as event-name pairs (id, label), where label is a descriptive keyword (e.g., leaked), and id identifies the resource triggering the event, and we use the notation $\mathcal{E}_{id}^{\mathsf{label}}$. Simulators and distinguishers can trigger events with arbitrary id's (looking forward, e.g. a simulator will have to trigger real-world events that do not occur in the ideal world). Still, we require that they do not trigger events that can be triggered by any resources they are connected to (such that, for example, a memory-leaked event really means that it did leak).

[1] From a conceptual point of view, this global event history is somewhat reminiscent of the "directory" ITI used in the recent version (as of December 2018) of UC [4] to keep track of which parties are corrupted.

Definition 3. *A simulator is* compatible *if it only triggers events that cannot be triggered by the resource it is attached to. For two resources* R *and* S*, a distinguisher* $\mathbf{D}$ *is* compatible *if it only triggers events that cannot be triggered by neither* R *nor* S*.*

Converters implementing protocols, on the other hand, do not depend on the event history, since an event is something that *might* be observable, rather than something that is guaranteed to be observable by the honest parties.

Construction notion. Intuitively, in the context of events, a real-world resource R is "just-as-good" as S if these resources look the same to distinguishers $\mathbf{D}^{\mathcal{E}}$ with read-and-write access to the global event history $\mathcal{E}$. This implies that the sequences of events must be the same in the real and in the ideal world. However, for convenience, we slightly relax this rule and introduce event renaming. For example, if a memory is used to store a key, then the memory-read event in the real world would have in the ideal world a better name key-received. Hence, we use both names to denote the same event (one can think of them as aliases). Moreover, we also allow for multiple aliases for a more fine-grained consideration of events in the ideal world, for instance by separating a message-received event into a successful and unsuccessful one.

We make this renaming explicit in the construction statements by defining a surjection τ that maps events triggered by the ideal-world resource to their real-world counterparts. (Note that in the case of duplicates caused by τ, $\tau(\mathcal{E})$ only contains the first occurrence.) When referring to real-world events for specifying ideal-world guarantees, we will sometimes use $\tilde{\mathcal{E}} \coloneqq \tau(\mathcal{E})$ as a shorthand notation.

We can now define the construction notion for two resources with events.

Definition 4. *We say that* $(\mathsf{prot}_1, \mathsf{prot}_2)$ *constructs* S *from* R *under the event-renaming* τ*, denoted*

$$\mathsf{prot}_1^{\mathsf{A}}\, \mathsf{prot}_2^{\mathsf{B}}\, \mathsf{R} \;\hat{\approx}_{\tau}\; \mathsf{sim}^{\mathsf{E}}\, \mathsf{S},$$

if there exists an efficient simulator sim*, such that* τ *only renames events triggered by* $\mathsf{sim}^{\mathsf{E}}\,\mathsf{S}$*, and for all efficient event-aware distinguishers* $\mathbf{D}^{\mathcal{E}}$*, compatible for* $\mathsf{prot}_1^{\mathsf{A}}\,\mathsf{prot}_2^{\mathsf{B}}$ *and* $\mathsf{sim}^{\mathsf{E}}\,\mathsf{S}$ *the following advantage is negligible.*

$$\Delta^{\mathbf{D}^{\mathcal{E}}}\big(\mathsf{prot}_1^{\mathsf{A}}\,\mathsf{prot}_2^{\mathsf{B}}\,\mathsf{R},\ \mathsf{sim}^{\mathsf{E}}\,\mathsf{S}\big) \coloneqq \Pr\big[\mathbf{D}^{\tau(\mathcal{E})}(\mathsf{sim}^{\mathsf{E}}\,\mathsf{S}) = 1\big] - \Pr\big[\mathbf{D}^{\mathcal{E}}(\mathsf{prot}_1^{\mathsf{A}}\,\mathsf{prot}_2^{\mathsf{B}}\,\mathsf{R}) = 1\big]$$

We stress that this construction notion satisfies the axioms of the more abstract layer on which the composition theorem of CC is proven [17, 18], and thus composes as well.

4 Composable Guarantees for Secure Messaging

In this section we introduce the unified type of construction statement—in CC with events—that we make about SM protocols and components thereof.

4.1 The Approach

We opt for the natural choice of an *application-centric approach*, where the security of a cryptographic scheme or primitive is defined as the construction it achieves when used in a particular application. While this approach provides readily understandable and clean security statements, the resulting definitions often turn out to be overly specific. For instance, the statement about an encryption scheme might hard-code a particular assumed authentic communication network, implying that it cannot be directly combined with an authentication scheme achieving slightly different guarantees.

Avoiding such overly specific statements is crucial for a modular treatment of ratcheting protocols, as each sub-protocol of the prior literature achieves slightly different guarantees. We address this problem by making parameterized construction statements, where the assumed real-world resources are parameterized by several "switches" determining their security guarantees. Formally, such a "switch" is represented by a function of the global event history $\mathcal{E}$ (among others), that dynamically defines the behavior of the resource at a given moment in time. For instance, a leakage function $\mathfrak{L}$ may specify to which extent a channel leaks depending on the set of events that happened so far. The goal of a protocol is then expressed as improving certain parameters while leaving the others unchanged, independently of what they were in the beginning. That is, our construction statements will be of the type that a protocol constructs a communication network with certain (stronger) guarantees, assuming a network with certain (weaker) guarantees, where the real-world guarantees are treated as a parameter instead of hard-coding them.

Note that in the context of ratcheting protocols, making such parameterized statement about components—without a-priori assuming any guarantees about the real-world—is mostly not an issue. This is due to the fact that the protocols anyway have to be designed for the setting where the state and randomness could leak at any time, temporarily nullifying all guarantees that the component might try to assume from the underlying sub-protocols.

4.2 Our Channel Model

We now introduce our model of two-party communication networks. It allows us to express flexible security guarantees, but also various usability restrictions or guarantees, such as whether messages can be received out of order or not.

Many single-message channels. We choose to model the communication network between Alice and Bob as the parallel composition of many unidirectional single-message communication channels. Besides being simpler to describe, it allows to have simpler construction steps which only consider a subset of the channels. On the flip side, it results in a world with an arbitrary but bounded number of messages, as the set of resources is static in CC. This is, however, without loss of generality as long as the protocols do not take advantage of this upper bound. Finally, observe that this decision results in a network where messages have implicit (unprotected) sequence numbers, as for instance achieved by TCP.

The single-message channel. We model channels with explicit authenticated data. Since we will use the same type of channel both in the real and ideal world, the channel must hit the right trade-off between giving enough power to the simulator but not too much power to the real-world adversary.

On a high level, the channel interfaces and their capabilities are as follows. See Figure 3 for the formal definition.

– The sender `S` can issue the command $(\texttt{send}, m, ad)$. Whether she is allowed to do so is determined by the can-send predicate $\mathfrak{S}$. (This predicate will mainly be used to describe situations in which the sender does not have the necessary keys yet.) A successful sending operation triggers the event $\mathcal{E}^{\mathsf{sent}}$. The sender can also query whether the channel is available for transmission.
– The adversary `E` can then potentially learn m through the `read` command. Whether she is allowed to do so is determined by the can-leak function $\mathfrak{L}$, which outputs either `false` (the adversary is not allowed to read m), `true` (reading is allowed but triggers a leaked event $\mathcal{E}^{\mathsf{leaked}}$), or `silent` (reading is allowed). Moreover, she is always allowed to learn the length of m and the (non-confidential) associated data ad.
– The adversary decides when receiving becomes possible, i.e., the message in principle is delivered. Once this happens, the receiver `R` can try to fetch the message. This has two possible outcomes: either he receives a message and an according received event is triggered, or he receives $\perp$ and an error event (indexed by an error code from Errors) is triggered. Which case happens is determined by the delivery function $\mathfrak{D}$, which takes into account the event history and on whether the message that `R` tries to fetch is the same as the one input by `S` (or an injected value from the adversary). The latter condition is denoted by the flag *same*. The flag *same* is also exposed as part of the received or error event $\mathcal{E}^{\mathsf{received}(same)}$ or $\mathcal{E}^{\mathsf{error}(err,same)}$, respectively.
– When the adversary decides that receiving is possible, she has two options: schedule the delivery of (m', ad') (command `deliver`), or force an error $err \in \mathsf{Errors}$ to be triggered (command `error`). In the first case, she can also request to just forward the sender's message (if one exists), using $m' = \texttt{fwd}$. Moreover, for technical reasons (often needed by the simulator), she can also insist that once the receiver fetches the message, $same = \texttt{false}$ is used even if the messages match. In case the adversary forces an error err and the outcome of receiving would anyway be a (different) error, the existing error can either be overwritten or preserved. She can control this by specifying a set *Overw* of errors that should be overwritten.

4.3 A Note on Confidentiality

In our channel, the $\mathcal{E}^{\mathsf{received}(same)}$ and $\mathcal{E}^{\mathsf{error}(err,same)}$ events indicate whether the message that Eve injected was the same as the sender's. Since we assume that those events are in principal observable by everybody, including the adversary, those events can partially breach confidentiality if the communication is not properly authenticated.

Resource $\mathsf{Ch}^{\mathsf{id},\mathtt{S}\to\mathtt{R}}_{\mathfrak{L},\mathfrak{I},\mathfrak{S},\mathfrak{R},\mathsf{Errors}}$

Parameters:
- Identity id (optionally), and interfaces $\mathtt{S}$ (sender) and $\mathtt{R}$ (receiver)
- Set of Errors that can occur
- Functions $\mathfrak{L}(\mathcal{E}) \in \{\mathtt{true}, \mathtt{false}, \mathtt{silent}\}$ (can leak), $\mathfrak{S}(\mathcal{E}) \in \{\mathtt{true}, \mathtt{false}\}$ (can send), $\mathfrak{R}(\mathcal{E}) \in \{\mathtt{true}, \mathtt{false}\}$ (can receive) and $\mathfrak{D}(\mathcal{E}, \mathit{same}) \in \mathsf{Errors} \cup \{\mathtt{msg}\}$ (delivery outcome)

Events: $\mathcal{E}^{\mathsf{sent}}$, $\mathcal{E}^{\mathsf{leaked}}$, $\mathcal{E}^{\mathsf{received}(\mathit{same})}$ and $\mathcal{E}^{\mathsf{error}(\mathit{err},\mathit{same})}$ for $\mathit{same} \in \{\mathtt{true}, \mathtt{false}\}$ and $\mathit{err} \in \mathsf{Errors}$

Initialization

$m_{\mathtt{S}}, ad_{\mathtt{S}}, cmd \leftarrow \bot$

Interface S

Input: $(\mathtt{send}, m, ad) \in \mathcal{M} \times \mathcal{AD}$
  assume $cmd = \bot$
  if $\neg\mathfrak{S}(\mathcal{E})$ **then output** $\bot$
  $(m_{\mathtt{S}}, ad_{\mathtt{S}}) \leftarrow (m, ad)$
  $\mathcal{E} \xleftarrow{+} \mathcal{E}^{\mathsf{sent}}$
  output `ok`

Input: `isAvailable`
  output $\mathfrak{S}(\mathcal{E})$

Interface R

Input: `receive`
  assume only called once
  if $\neg\mathfrak{R}(\mathcal{E}) \vee cmd = \bot$ **then**
    output $\bot$

  // same messages (no injection)?
  if $\mathit{same} = \mathtt{check}$ **then**
    $\mathit{same} \leftarrow ((m_{\mathtt{R}}, ad_{\mathtt{R}}) = (m_{\mathtt{S}}, ad_{\mathtt{S}}))$

  // the outcome: an error or the message
  $out \leftarrow \mathfrak{D}(\mathcal{E}, \mathit{same})$
  if $cmd = \mathtt{dlv} \wedge out = \mathtt{msg}$ **then**
    $\mathcal{E} \xleftarrow{+} \mathcal{E}^{\mathsf{received}(\mathit{same})}$
    output $(m_{\mathtt{R}}, ad_{\mathtt{R}})$
  else if $cmd = (\mathtt{err}, \mathit{err}, \mathit{Overw})$
      $\wedge (out = \mathtt{msg} \vee out \in \mathit{Overw})$ **then**
    $\mathcal{E} \xleftarrow{+} \mathcal{E}^{\mathsf{error}(\mathit{err},\mathit{same})}$
    output $\bot$
  else
    $\mathcal{E} \xleftarrow{+} \mathcal{E}^{\mathsf{error}(\mathit{out},\mathit{same})}$
    output $\bot$

Input: `isAvailable`
  output $\mathfrak{R}(\mathcal{E}) \wedge cmd \neq \bot$

Interface E

Input: `read`
  if $\mathfrak{L}(\mathcal{E}) = \mathtt{false}$ **then output** $\bot$
  else if $\mathfrak{L}(\mathcal{E}) = \mathtt{true}$ **then** $\mathcal{E} \xleftarrow{+} \mathcal{E}^{\mathsf{leaked}}$
  output $(m_{\mathtt{S}}, ad_{\mathtt{S}})$

Input: `readLength`
  output $(|m_{\mathtt{S}}|, ad_{\mathtt{S}})$

Input: $(\mathtt{deliver}, m, ad, \mathit{same}')$
    $\in (\mathcal{M} \cup \{\mathtt{fwd}\}) \times \mathcal{AD} \times \{\mathtt{check}, \mathtt{false}\}$
  assume $cmd = \bot$

  // handle forwarding request
  if $m = \mathtt{fwd}$ **then**
    if $m_{\mathtt{S}} = \bot$ **then output** $\bot$
    else $m \leftarrow m_{\mathtt{S}}$

  // store for receiving
  $(m_{\mathtt{R}}, ad_{\mathtt{R}}, \mathit{same}) \leftarrow (m, ad, \mathit{same}')$
  $cmd \leftarrow \mathtt{dlv}$
  output `ok`

Input: $(\mathtt{error}, \mathit{err}, \mathit{Overw}, m, ad, \mathit{same}')$
    $\in \mathsf{Errors} \times 2^{\mathsf{Errors}} \times \mathcal{M} \times \mathcal{AD}$
    $\times \{\mathtt{true}, \mathtt{false}, \mathtt{check}\}$
  assume $cmd = \bot$

  // (m, ad) only to determine same
  if $\mathit{same}' = \mathtt{check}$ **then**
    $(m_{\mathtt{R}}, ad_{\mathtt{R}}) \leftarrow (m, ad)$

  // store for receiving
  $\mathit{same} \leftarrow \mathit{same}'$
  $cmd \leftarrow (\mathtt{err}, \mathit{err}, \mathit{Overw})$
  output `ok`

Fig. 3: The single-message channel.

However, those events are crucial to phrase the post-impersonation guarantees of of certain ratcheting protocols. In fact, in those protocols Eve could usually inject her own message (after exposing the sender's state), observe whether it causes the communication to break down, and thereby deducing whether the sender wanted to send the same message afterwards. Our events simply reflect this.

4.4 Additional Resources: Memory and Randomness

An integral part of secure messaging protocols is the assumption that the parties' state, and sometimes also randomness, can leak to the adversary. In Constructive Cryptography everything that can be accessible by multiple parties, here the honest party and Eve, must be modeled as a resource. As a consequence, all of our converters will be stateless and deterministic. (Stateless means that the converter cannot keep state between two separate invocations at the emulated interfaces.) The statements will contain explicit memory and randomness resources instead. These resources are formally defined in Figure 4.

On a high level, we consider two types of memory resources: (1) an insecure memory $\mathsf{IMem}^{id,\mathtt{U}}$, and (2) a potentially secure memory $\mathsf{Mem}^{id,\mathtt{U}}$. The former is multiple-use and its current content is always available to the adversary. On the other hand, a secure memory can be written to at most once. It can also be securely erased at any later time. Moreover, it is parameterized by a can-leak predicate $\mathfrak{L}$, that specifies whether the content is available to the adversary. When the adversary successfully reads the contents, a leaked event is triggered.[2] Observe that each memory can leak independently, which leads to more fine-grained statements compared to prior work where it was usually assumed that either the entire state leaks or not (a state often consists of many secret keys from different sub-protocols, which we put in different memories). Nevertheless, it does not appear to incur additional significant complications.

Defining a potentially leakable randomness resource is a bit subtle. In principle, the idea is that the randomness can leak to the adversary *at the moment* it is used (modeling that it is sampled fresh at this point and is not stored) by the honest party. However, this cannot be directly expressed like this due to the activation model of the version of Constructive Cryptography used (recall that the output is given at the same interface the input was given). Hence, we model randomness resources that can be in one of two states: leakable or not (as specified by the flag *leaks*). If the can-leak predicate evaluates to true, the adversary can switch the state to leakable by sending `triggerLeaking`, which also triggers the leaked event. When the resource is used by the honest party, fresh randomness is sampled. Additionally, if at this time the state is leakable, then the sampled value is stored and the adversary can read it at any time afterwards.

5 Unifying Ratcheting: Two Examples

In this section, we get acquainted with how the security guarantees of ratcheting protocols can be phrased within our model. To this end, we model the guarantees of two components of actual ratcheting protocols.

[2] Rewritable secure memory can then be modeled as the parallel composition of many write-once memory cells. The memory requirement of a protocol is not determined by the number of such write-once memories, but rather by the maximal number of them in use at any time.

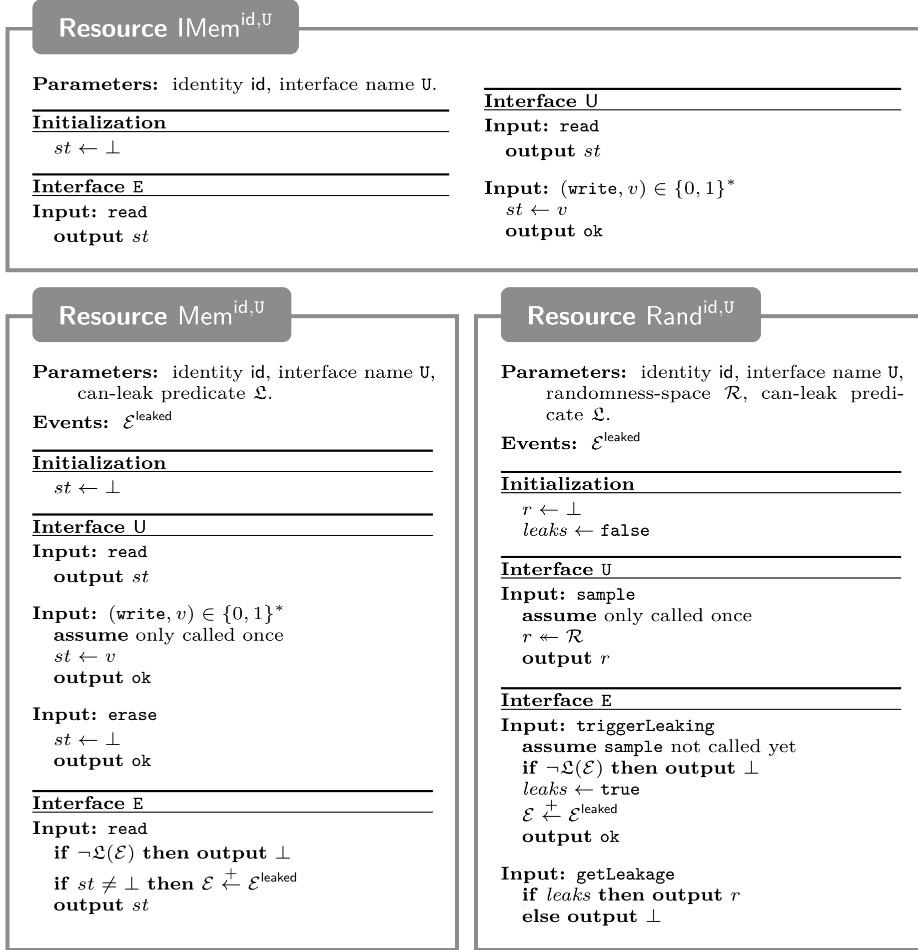

Fig. 4: Formal definition of the memory and randomness resources.

As a first example, we consider a simple authentication scheme that appears in [9, 8, 10]. Using this example, we demonstrate how our framework allows for a fine-grained modularization, with the overall security then directly following from composition. As a second example, we consider the use of hierarchical identity-based encryption, as in [23, 9]. In this example, we explore a way to work around the so-called commitment issue of composable security.

5.1 A Simple Authentication Scheme

We first consider a simple unidirectional authentication protocol, which is designed with the strong guarantees of secure messaging in mind: the authentication guarantees should not only be forward secure but also heal after a state or

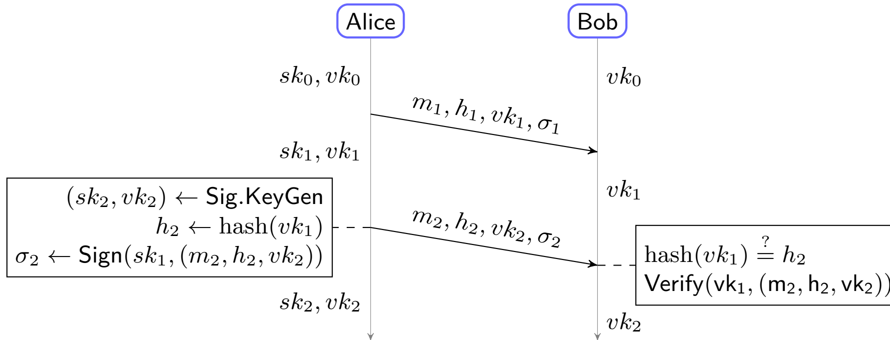

Fig. 5: The simple scheme for unidirectional authentication.

randomness exposure of either party. Slight variations of this protocol have been used in [10] and [8]. Essentially the same idea also appeared in [9], where, however, a stronger signature primitive with updatable keys is considered, leading to the protocol being formalized in the bidirectional setting.

The protocol. In the protocol, whenever the sender wants to send a message, a fresh signing and verification key pair is sampled. The fresh verification key is then signed together with the message—using the prior signing key— and the message, the verification key and the signature are transmitted. Finally, the old signing key is securely erased and the fresh one stored instead. The receiver, on the other hand verifies a received message with the previous verification key and stores the new one. The scheme is depicted in Figure 5.

Recall that we aim to make a strong construction statement that considers how the scheme enhances any preexisting security guarantees, including confidentiality. Usually preserving confidentiality is not a goal that is considered for an authentication protocol, moreover, it is known that the authenticate-then-encrypt approach used in old versions of TLS is not generally secure [12]. Nevertheless, we show that the scheme actually achieves this at the cost of assuming unique signatures instead of unforgeable ones (analogous to [9]), and with a minor modification: with each message, the sender also transmits a hash of the previous verification key. Such a hash is also present in the protocol from [9], and allows the receiver to check whether he is using the correct verification key.

The guarantees. Clearly, the protocol achieves authenticity if neither party's state is exposed. Moreover, Bob's state only consists of public information. If Alice's state gets exposed, then Eve obtains her current signing key that she can use to impersonate Alice towards Bob at this point in time. However, this key is useless to tamper previous messages, even if they have not been delivered yet (forward security). More importantly, if, for some reason, Alice's next message containing a fresh verification key still is delivered without modification, then the signing key obtained by the adversary becomes useless thereby achieving the

healing property. Hence, the adversary can inject the i-th message if and only if Alice's state between the $(i-1)$-st and i-th message got exposed, or there has already been a successful injection before.

Expressing the scheme's security guarantees in a game-based manner turned out to be surprisingly involved compared to the scheme's simplicity and how easy it seems to intuitively describe its guarantees. Notably, to show its security, in [10] the abstraction of a key-updating signature scheme, as well as its corresponding correctness and security games, have been introduced. See Appendix A.1 for the corresponding definitions. This raises a couple of questions: can't we do simpler? What is the right security statement to make about this quite simple protocol, and what happens if the channel already provides certain authenticity or confidentiality guarantees? In the following, we try to answer these questions.

The construction statement. First, note that we consider the authentication of messages directly, and do not introduce an intermediate signature notion. Secondly, we consider authenticating the i-th message only, and to this end consider the $(i-1)$-st message where the fresh verification key is transmitted (we do not authenticate this message here) and the i-th message that is then signed under the corresponding signing key. Authenticating the $(i-1)$-st message, and all others, is then taken care of by iteratively applying the protocol, with the overall security directly implied by the composition theorem. This leads to the following real world resources

$$\mathsf{R}_i^{\mathsf{auth}} := \left[\mathsf{Ch}^{i-1,\mathtt{A}\to\mathtt{B}}, \mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}, \mathsf{Rand}^{kg_i,\mathtt{A}}, \mathsf{Mem}^{sk_i,\mathtt{A}}, \mathsf{IMem}^{vk_i,\mathtt{B}}\right], \tag{1}$$

where besides the two channels the sender also has a memory to store the new signing key, and the receiver a (insecure) memory to store the verification key. Furthermore, the sender also has an explicit randomness resource available (note that we only need key-generation randomness, since unique signatures are deterministic). The corresponding protocol converters $(\mathsf{sig}_i, \mathsf{vrf}_i)$ that are connected to Alice's and Bob's interfaces of $\mathsf{R}_i^{\mathsf{auth}}$, respectively, simply implement the previously described protocol. A formal description of those protocol converters can be found in Appendix A.2.

The goal of the protocol is then phrased as constructing the following ideal-world resource

$$\mathsf{S}_i^{\mathsf{auth}} := \left[\mathsf{Ch}^{i-1,\mathtt{A}\to\mathtt{B}}, \mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}\right], \tag{2}$$

in which the channels can also trigger an error $\mathsf{sig\text{-}err}$, indicating that the signature verification failed, in addition to the errors from the real-world counterparts.

The authentication guarantees for the i-th channel can then be expressed via the following delivery-function, which guarantees that an injection attempt $(\neg same)$ when the key is not known will causes a signature-verification error $\mathsf{sig\text{-}err}$, and preserves preexisting authenticity (recall that $\tilde{\mathcal{E}} := \tau(\mathcal{E})$ denotes the

real-world's event history):

$$\mathfrak{D}^{\mathsf{S}_i^{\mathsf{auth}}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}(\mathcal{E}, \mathit{same}) \coloneqq \begin{cases} \mathit{err} & \text{if } \mathfrak{D}^{\mathsf{R}_i^{\mathsf{auth}}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}(\tilde{\mathcal{E}}, \mathit{same}) = \mathit{err} \wedge \mathit{err} \neq \mathtt{msg} \\ \mathtt{msg} & \text{else if } \mathit{same} \vee \mathcal{E}_i^{\mathsf{sk\text{-}known}} \\ \mathsf{sig\text{-}err} & \text{else} \end{cases} \tag{3}$$

where in a slight abuse of notation, we define a composed event $\mathcal{E}_i^{\mathsf{sk\text{-}known}}$, which is triggered as soon as it is not excluded that the signing key corresponding to Bob's verification key is known to Eve:

$$\mathcal{E}_i^{\mathsf{sk\text{-}known}} \coloneqq \mathcal{E}^{\mathsf{injected}}_{\mathsf{Ch}(i-1,\mathtt{A}\to\mathtt{B})} \vee \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Rnd}(kg_i,\mathtt{A})} \vee \left(\mathcal{E}^{\mathsf{sent}}_{\mathsf{Ch}(i-1,\mathtt{A}\to\mathtt{B})} \prec \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Mem}(sk_i,\mathtt{A})} \prec \mathcal{E}^{\mathsf{sent}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}\right).$$

On the flip side, the scheme limits the availability of the channels to be sequential. While sending messages in order is natural for Alice, the protocol restricts Bob to receive them in order as well. We can express this using the following predicates.

$$\mathfrak{S}^{\mathsf{S}_i^{\mathsf{auth}}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}(\mathcal{E}) \coloneqq \mathfrak{S}^{\mathsf{R}_i^{\mathsf{auth}}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}(\tilde{\mathcal{E}}) \ \wedge\ \mathcal{E}^{\mathsf{sent}}_{\mathsf{Ch}(i-1,\mathtt{A}\to\mathtt{B})}, \tag{4}$$

$$\mathfrak{R}^{\mathsf{S}_i^{\mathsf{auth}}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}(\mathcal{E}) \coloneqq \mathfrak{R}^{\mathsf{R}_i^{\mathsf{auth}}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}(\tilde{\mathcal{E}}) \ \wedge\ \mathcal{E}^{\mathsf{received}}_{\mathsf{Ch}(i-1,\mathtt{A}\to\mathtt{B})}. \tag{5}$$

Note that our model simply forces us to make this restriction explicit, whereas this is often just hard-coded in games.[3]

All other parameters and predicates are preserved, e.g. $\mathfrak{L}^{\mathsf{S}_i^{\mathsf{auth}}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}(\mathcal{E}) \coloneqq \mathfrak{L}^{\mathsf{R}_i^{\mathsf{auth}}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}(\tilde{\mathcal{E}})$. The security of the protocol can then be phrased as constructing the ideal world $\mathsf{S}_i^{\mathsf{auth}}$ from the real world $\mathsf{R}_i^{\mathsf{auth}}$, as summarized in the following theorem.

Theorem 1. *Let $\mathsf{R}_i^{\mathsf{auth}}$ be as in (1), and let $\mathsf{S}_i^{\mathsf{auth}}$ be as in (2), with the guarantees and restrictions as described in (3), (4), and (5), respectively, and all others guarantees unchanged from $\mathsf{R}_i^{\mathsf{auth}}$. Moreover, let τ map the event $\mathcal{E}^{\mathsf{error}(\mathsf{sig\text{-}err},\mathit{same})}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}$ to $\mathcal{E}^{\mathsf{received}(\mathit{same})}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}$. Then there exists an efficient simulator sim such that*

$$\mathsf{sig}_i^{\mathsf{A}}\ \mathsf{vrf}_i^{\mathsf{B}}\ \mathsf{R}_i^{\mathsf{auth}}\ \approx_\tau\ \mathsf{sim}^{\mathsf{E}}\ \mathsf{S}_i^{\mathsf{auth}},$$

if the underlying signature scheme is unforgeable with unique signatures, and the hash function is collision resistant.

Proof. The proof is found in Appendix A.3. Note that compared to a normal signature-scheme proof it is quite involved, which is the main price we pay for our much stronger statement.

[3] Actually, many recently proposed secure-messaging protocols do have this restriction, which might limit their usability as pointed out by [1].

Extending to many messages. So far, we only considered a world where Alice sends two messages, of which the second is authenticated. In a realistic setting, Alice can of course send many messages where all of them should be authenticated. In this section, we see how the composition theorem of Constructive Cryptography can be applied to directly get the desired result.

In particular, we start with a sequence of possibly unauthenticated channels $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$ for $i \in [n]$, where the authentication of $\mathsf{Ch}^{0,\mathtt{A}\to\mathtt{B}}$ can be seen as a setup assumption (it is standard to assume that Alice and Bob initially share a signing-verification key pair). Then, we iteratively apply the construction for two channels to $\mathsf{Ch}^{0,\mathtt{A}\to\mathtt{B}}$ and $\mathsf{Ch}^{1,\mathtt{A}\to\mathtt{B}}$, then to $\mathsf{Ch}^{1,\mathtt{A}\to\mathtt{B}}$ and $\mathsf{Ch}^{2,\mathtt{A}\to\mathtt{B}}$, etc. (c.f. Figure 6). The composition theorem of CC guarantees that the composed protocol constructs the ideal world.

Corollary 1. *Let* $\mathsf{R}^{\mathsf{auth}}$ *and* $\mathsf{S}^{\mathsf{auth}}$ *denote the following real and ideal worlds*

$$\mathsf{R}^{\mathsf{auth}} := \left[\left\{\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}\right\}_{i\in\{0,\dots,n\}}, \left\{\mathsf{Mem}^{sk_i,\mathtt{A}}, \mathsf{IMem}^{vk_i,\mathtt{B}}\right\}_{i\in[n]}\right],$$

and

$$\mathsf{S}^{\mathsf{auth}} := \left[\left\{\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}\right\}_{i\in\{0,\dots,n\}}\right],$$

respectively. Then, there exists an efficient simulator sim *such that*

$$(\mathsf{sig}_1,\dots,\mathsf{sig}_n)^{\mathtt{A}}\ (\mathsf{vrf}_1,\dots,\mathsf{vrf}_n)^{\mathtt{B}}\ \mathsf{R}^{\mathsf{auth}} \ \approx\ \mathsf{sim}^{\mathtt{E}}\mathsf{S}^{\mathsf{auth}},$$

where for each $i \in [n]$, $\mathfrak{I}^{\mathsf{S}^{\mathsf{auth}}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}$, $\mathfrak{S}^{\mathsf{S}^{\mathsf{auth}}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}$, *and* $\mathfrak{R}^{\mathsf{S}^{\mathsf{auth}}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}$ *are defined as in* (3), (4), *and* (5), *respectively.*

5.2 Confidentiality from HIBE

In the following we discuss a protocol from [9] that uses *hierarchical identity-based encryption (HIBE)* to add confidentiality to a sequence of channels. The protocol was designed for a challenging setting, where we do not assume authentication (as is usually done when talking about encryption). The reason is that in secure messaging authentication cannot be guaranteed when the sender's state is exposed. This situation fits perfectly to our framework.

The protocol is described in the so-called sesqui-directional setting, introduced in [23], meaning that the messages from both directions are considered, but only the guarantees of one of the directions are under concern—here from Alice to Bob. The bidirectional guarantees then follow directly from composition.

Hierarchical identity-based encryption. A HIBE scheme consists of the following four algorithms:

- A setup generation algorithm $(mpk, msk) \leftarrow \text{HIBE.Setup}(1^\kappa; r)$, generating the root master public and secret keys, i.e. $sk_{()} = msk$.

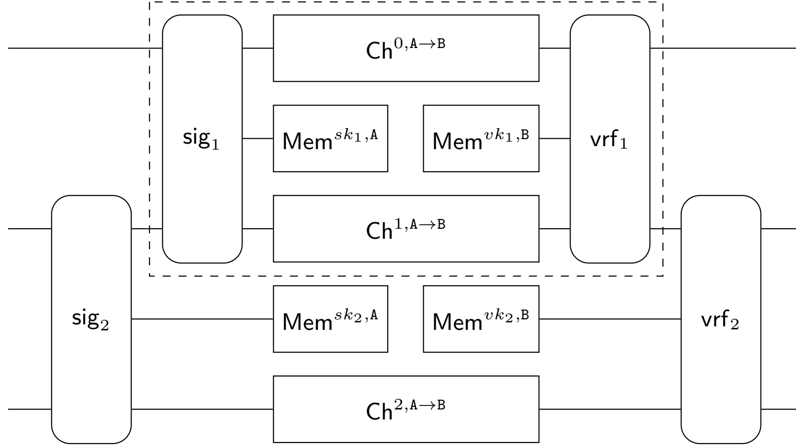

Fig. 6: The first two steps constructing a sequence of authenticated channels: (1) The protocol $(\mathsf{sig}_1, \mathsf{vrf}_1)$ constructs a hybrid world, where the resources in the dashed box are replaced by two channels $\mathsf{Ch}^{0,\mathtt{A}\to\mathtt{B}}$ and $\mathsf{Ch}^{1,\mathtt{A}\to\mathtt{B}}$, where $\mathsf{Ch}^{1,\mathtt{A}\to\mathtt{B}}$ is authenticated as long as $\mathsf{Ch}^{0,\mathtt{A}\to\mathtt{B}}$ is. (2) $(\mathsf{sig}_2, \mathsf{vrf}_2)$ constructs the ideal world, where $\mathsf{Ch}^{1,\mathtt{A}\to\mathtt{B}}$ and $\mathsf{Ch}^{2,\mathtt{A}\to\mathtt{B}}$ are authenticated as long as $\mathsf{Ch}^{0,\mathtt{A}\to\mathtt{B}}$ is.

- A key-generation algorithm $sk_{\mathrm{id}\|id_n} \leftarrow \mathrm{HIBE.Kgen}(sk_{\mathbf{id}}, id_n)$, where $(\mathbf{id} \| id_n) \coloneqq (id_1, \ldots, id_{n-1}, id_n)$ for an identity vector $\mathbf{id} = (id_1, \ldots, id_{n-1})$.
- An encryption algorithm $c \leftarrow \mathrm{HIBE.Enc}(mpk, \mathbf{id}, m; r)$.
- A decryption algorithm $m \leftarrow \mathrm{HIBE.Dec}(sk_{\mathbf{id}}, c)$.

We require the HIBE scheme to be IND-CCA secure with certain additional properties that are not guaranteed by IND-CCA itself, but that most schemes do provide (see Appendix B.2 for details).

The protocol overview. On a high level, the protocol proceeds in epochs, where in each epoch Bob sends one message to Alice, and then Alice sends a sequence of messages to Bob. In particular, Bob's message contains a fresh HIBE public key *mpk*. For simplicity, consider the first epoch, as depicted in Figure 7. When Alice sends her i-th message, she encrypts it with *mpk*, using as the identity (the hashes of) all ciphertexts she sent before. Whenever Bob receives a ciphertext c_i, he decrypts it, derives the secret key for the new identity (with c_i appended) and erases the old key.

In the next epoch, Bob sends a new public key mpk', and we repeat. One subtle issue is how to run the epochs together. Note that, for example, Bob may send a number of public keys without receiving a response, in which case he has to store secret keys from a number of epochs. A fresh secret key is stored for the empty identity, and when Bob receives a ciphertext, he updates all currently stored secret keys. This means that Alice uses for encryption of the i-th message a truncated transcript $(c_r, \ldots, c_{i-1})$. In order for her to compute it, Bob sends

with each public key the index r of the last message he received. A graphical depiction of the full protocol can be found in Appendix B.1.

Security intuition. Intuitively, this use of HIBE allows to achieve three goals. The first is healing, achieved by exchanging fresh keys, as in most secure-messaging schemes. The second is forward secrecy: exposing the secret key after the i-th message is received does not affect the confidentiality of messages $m_1, \dots, m_{i-1}$. This holds, since Bob updated all the secret keys with the identity c_i in the meantime. Healing and forward secrecy could also be achieved by a forward-secure PKE scheme. The last goal is the so-called post-impersonation security: an active injection destroys the decryption keys, so that its leakage exposes no messages. For this we need the hierarchy of identities. Roughly, injecting a message c_i' causes Bob to update his key to $sk_{(c_r,\dots,c_i')}$. This key gives no information about messages encrypted by Alice, since those will be for another identity $(c_r, \dots, c_i)$.

The construction statement. To formalize these guarantees as a construction statement, we first have to describe the real world in which the protocol is executed. It consists of n channels from Alice to Bob (which the protocol protects) and n channels in the opposite direction on which the master public keys are transmitted. Moreover, Alice has memories to store the public keys and the transcript, and randomness resources for the encryption. Bob, on the other hand, has memories to store the secret keys and randomness resources for the key generation:

$$\mathsf{R}^{\mathsf{hibe}} := \Big[\big\{\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}\big\}_{i\in[n]}, \big\{\mathsf{Ch}^{j,\mathtt{B}\to\mathtt{A}}\big\}_{j\in[n]}, \mathsf{IMem}^{pk,\mathtt{A}}, \big\{\mathsf{Rand}^{kg_j,\mathtt{B}}\big\}_{j\in[n]}, \big\{\mathsf{Mem}^{tr_i,\mathtt{A}}, \mathsf{Rand}^{enc_i,\mathtt{A}}\big\}_{i\in[n]}, \big\{\mathsf{Mem}^{sk_{(j,i)},\mathtt{B}}\big\}_{j\in[n],i\in[n+1]\}}\Big], \quad (6)$$

where the index i indicates that the resource is related to transmitting the i-th message from Alice to Bob, and the index j indicates the j-th epoch. A formal description of the pair of converters implementing the protocol ($\mathsf{hibe\text{-}enc}$, $\mathsf{hibe\text{-}dec}$) can be found in Appendix B.1.

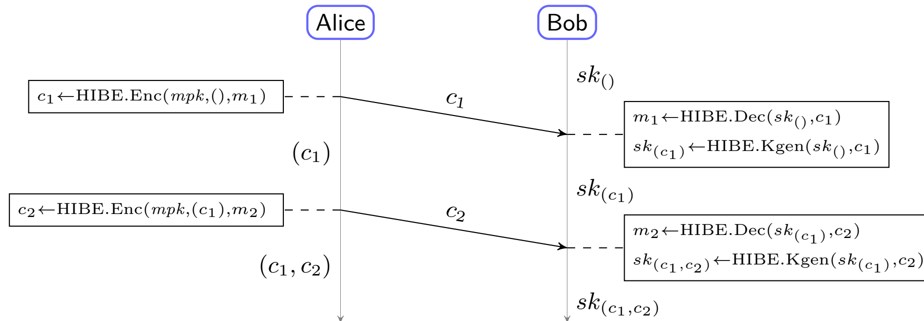

Fig. 7: The first epoch of the sesquidirectional HIBE protocol.

The goal of the protocol is to enhance the confidentiality of the channels. Thus, the same set of channels is present in the ideal world, while the memory and randomness resources are used up:

$$\mathsf{S}^{\mathsf{hibe}} := \Big[\big\{\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}\big\}_{i\in[n]}, \big\{\mathsf{Ch}^{j,\mathtt{B}\to\mathtt{A}}\big\}_{j\in[n]}\Big]. \tag{7}$$

Moreover, the ideal channels can trigger an additional error $\mathsf{dec\text{-}err}$, indicating that decryption failed (this error event corresponds to the real-world delivery event when the adversary injects an invalid ciphertext).

We now proceed to formalize the confidentiality guarantees of $\mathsf{S}^{\mathsf{hibe}}$ by defining in which situations the i-th message might be known to the adversary:

The randomness leaked: If the encryption randomness leaked to the adversary, i.e., $\mathcal{E}^{\mathsf{leaked}}_{\mathsf{Rnd}(enc_i,\mathtt{A})}$, then no PKE scheme can provide (full) confidentiality.

The master public key was set by Eve: If Alice encrypts using a master public key (potentially) set by Eve, Eve can trivially decrypt. That is, if Alice used the j-th master public key and $\mathcal{E}^{\mathsf{injected}}_{\mathsf{Ch}(j,\mathtt{B}\to\mathtt{A})}$.

The secret key leaked: Assume Alice sent the i-th message during the j-th epoch, and let $sk_{(j,i)}$ denote the secret key that Bob uses to decrypt that message. If Eve learned $sk_{(j,i)}$, the message is obviously not confidential, which either happens if the randomness used to generate the master secret key leaked or a key that allows to compute $sk_{(j,i)}$ leaked from Bob's memory:

$$\mathcal{E}^{\mathsf{sk\text{-}leaked}}_{i,j} := \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Rnd}(kg_j,\mathtt{B})} \\ \vee \exists k \in [r_j, i] : \Big(\mathcal{E}^{\mathsf{leaked}}_{\mathsf{Mem}(sk_{(j,k)},\mathtt{B})} \wedge \forall \ell \in [r_j, k] : \neg \mathcal{E}^{\mathsf{injected}}_{\mathsf{Ch}(\ell,\mathtt{A}\to\mathtt{B})}\Big),$$

where r_j denotes the first message Bob received after sending the j-th public key (r_j is determined by the sent and received events in $\mathcal{E}$). Note that the last condition explicitly encodes the *post-impersonation guarantee*, meaning that $sk_{(j,k)}$ is only useful as long as Eve did not destroy it by injecting her own ciphertext. Forward-secrecy and healing, on the other hand, are encoded implicitly by the order in which those events can happen in the real world. We can make them more explicit by observing

$$\mathcal{E}^{\mathsf{sk\text{-}leaked}}_{i,j} \iff \mathcal{E}^{\mathsf{sent}}_{\mathsf{Ch}(j,\mathtt{B}\to\mathtt{A})} \prec \mathcal{E}^{\mathsf{sk\text{-}leaked}}_{i,j} \prec \mathcal{E}^{\mathsf{received}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})},$$

where the former condition denotes healing and the latter forward-secrecy.

In summary, we can define the following event denoting that the i-th message is insecure

$$\mathcal{E}^{\mathsf{exposed}}_{i} := \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Rnd}(enc_i,\mathtt{A})} \vee \mathcal{E}^{\mathsf{injected}}_{\mathsf{Ch}(j_i,\mathtt{B}\to\mathtt{A})} \vee \mathcal{E}^{\mathsf{sk\text{-}leaked}}_{i,j_i},$$

where j_i denotes the epoch in which the i-th message has been sent (which is computable from the order of events in $\mathcal{E}$), leading to

$$\mathfrak{L}^{\mathsf{S}^{\mathsf{hibe}}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}(\mathcal{E}) := \begin{cases} \mathtt{silent} & \text{if } \mathcal{E}^{\mathsf{exposed}}_{i} \\ \mathtt{false} & \text{otherwise.} \end{cases} \tag{8}$$

Notice that the above can-leak function fully overwrites any real-world guarantees, and silences the leaked events. This is because in the protocol Alice stores the communication transcript. As a consequence, when her memory leaks, the ciphertext leaks as well, even if the assumed channel was in fact confidential. Moreover, this leakage does not correspond to the channel leaked event.

Analogous to the authentication scheme of the previous section, the HIBE scheme also limits the availability of the channels to be sequential, due to the hash-transcript used as identities. Moreover, Alice can obviously only encrypt using master public keys she received the public key. This could be made formal using the can-send and can-receive predicates $\mathfrak{S}$ and $\mathfrak{R}$, respectively, analogous to the previous section.

Working around the commitment problem. As described so far, the real and ideal world $\mathsf{hibe\text{-}enc}^{\mathsf{A}}\mathsf{hibe\text{-}dec}^{\mathsf{B}}\mathsf{R}^{\mathsf{hibe}}$ and $\mathsf{sim}^{\mathsf{E}}\mathsf{S}^{\mathsf{hibe}}$, respectively, are easily distinguishable for any simulator sim. The issue is the so-called *commitment problem* of simulation based cryptography: if the distinguisher chooses to first see a ciphertext and then leak the corresponding decryption key, this cannot be simulated, since the simulator first has to output a fake ciphertext, before getting to know the message, and then explain it by outputting a corresponding decryption key. For normal PKE, and especially HIBE, schemes this is impossible.

One solution would be to consider static memory corruptions, where the set of states that can be leaked to the adversary is a parameter of the construction statement. Such a static guarantee is however weaker than the existing game-based definitions and, thus, thwarts our goal of developing a unified model to express the guarantees obtained by existing protocols. We thus opt for the alternative solution to strengthen the real world analogous to how the games disable certain oracles to prevent trivial impossibilities. To this end, we disallow the adversary from obtaining the secret key $sk_{(j,i)}$ if this would allow to trivially identify a fake ciphertext. That is, we assume

$$\mathfrak{L}^{\mathsf{R}^{\mathsf{hibe}}}_{\mathsf{Mem}(sk_{(j,i)},\mathtt{B})}(\mathcal{E}) :\equiv \neg\exists k > i : \left(\mathcal{E}^{\mathsf{committed}}_{k,j} \prec \mathcal{E}^{\mathsf{exposed}}_{k}\right), \tag{9}$$

where $\mathcal{E}^{\mathsf{committed}}_{i,j}$ denotes the event that the simulator commits on the i-th ciphertext, and that it was encrypted under mpk_j. More concretely, this happens if the distinguisher

- explicitly asked for the ciphertext;
- requested a hash-transcript that depends on the ciphertext;
- requested a secret key for which the identity depends on the ciphertext;
- actively injected a ciphertext that got decrypted under a secret key whose identity depends on the ciphertext under consideration,

leading to the following definition

$$\mathcal{E}^{\mathsf{committed}}_{i,j} :\equiv (j_i = j) \land \Big(\mathcal{E}^{\mathsf{leaked}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})} \lor \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Mem}(tr_i,\mathtt{A})} \lor \Big(\neg\mathcal{E}^{\mathsf{injected}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})} \land \exists k \geq i : \left(\mathcal{E}^{\mathsf{leaked}}_{\mathsf{Mem}(sk_{(j,k)})} \lor \mathcal{E}^{\mathsf{injected}}_{\mathsf{Ch}(k,\mathtt{A}\to\mathtt{B})}\right)\Big)\Big),$$

where again j_i denotes the epoch in which the i-th message has been sent.

While the construction statement loses its evident executional semantics making those restrictions of the real world—it is no longer apparent what guarantees one gets when executing the protocol in the actual world where the memory leakage is obviously not restricted like this—it is analogous to game-based notions where the adversary has to chooses beforehand whether a message is a challenge (and then prevents leaking the corresponding randomness or secret keys), or is an insecure message just to advance the state. Phrasing it in a composable framework, however, still has the advantage of modularity and reusability, that is, each subprotocol can be proven secure independently and the overall security directly following from the composition theorem.

Summary and analysis. The HIBE-based scheme achieves the so far described construction, with one exception: to provide more power to the simulator and make the construction statement provable, we need to silence the real-world channels' leaked events after the message is exposed, i.e, $\mathfrak{L}^{\mathsf{R}^{\mathsf{hibe}}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}$ is arbitrary, except that if $\mathcal{E}^{\mathsf{exposed}}_{i}$, it no longer evaluates to `true`.[4]

Observe that while having to silence the leakage event in the real world limits reusability, the statement for instance is still generic enough to be composed with the authentication scheme from the previous section: if the real world is restricted like this (in the end, those events are just a mean to phrase dependencies and carry no real semantics), then the signature scheme, which preserves the can-leak predicate, and afterwards the HIBE scheme can be applied.

Overall, we have the following theorem, proved in Appendix B.2.

Theorem 2. *Let $\mathsf{R}^{\mathsf{hibe}}$ be as in (6) with the restrictions to work around the commitment-problem from (9) and the restriction described above, and let $\mathsf{S}^{\mathsf{hibe}}$ be as in (7) with the confidentiality guarantees from (8), and in-order sending and receiving. Let τ map the event $\mathcal{E}^{\mathsf{error}(\mathsf{dec\text{-}err},same)}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}$ to $\mathcal{E}^{\mathsf{received}(same)}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}$. Then there exists an efficient simulator sim, such that*

$$\mathsf{hibe\text{-}enc}^{\mathtt{A}}\,\mathsf{hibe\text{-}dec}^{\mathtt{B}}\,\mathsf{R}^{\mathsf{hibe}} \;\approx_{\tau}\; \mathsf{sim}^{\mathtt{E}}\,\mathsf{S}^{\mathsf{hibe}},$$

if the HIBE scheme is IND-CCA secure with our additional assumptions.

6 Adaptive Security

All protocols considered so far, and most of the ones in the literature, only achieve a weakened construction statement, where, due to the commitment problem, we assume that certain sequences of events cannot occur in the real world. Intuitively, this means that the adversary is somewhat static: for example, when she decides to see the contents of a channel (the ciphertext, in the real world), at the same time she decides that she will not look at the contents of certain memory (the secret

[4] This doesn't affect $\mathcal{E}^{\mathsf{committed}}_{i,j}$, that only considers leakage events before $\mathcal{E}^{\mathsf{exposed}}_{i}$.

key). While this is exactly what the standard game-based definitions guarantee, when expressed in a composable framework, it seems rather unsatisfactory.

Hence, in this section, we consider SM schemes that tolerate a fully adaptive adversary, i.e, allow to "explain" ciphertexts whenever needed due to leakage of secret keys. In particular, we present a technique that, given an SM protocol that suffers from the commitment problem, allows to construct an adaptive SM (ASM) protocol with almost the same guarantees, but that achieves fully adaptive security. This comes at the cost of efficiency and being able to send only a fixed number of messages without interaction. Applied to protocols with optimal security [9, 23], our technique enables even stronger guarantees.[5] As an example, we apply it to the HIBE protocol from Section 5.2.

Note that while the technique we use is essentially a general compiler that "removes" the commitment problem, formally phrasing such a theorem would be rather cumbersome for at least two reasons. First, there is not just one game-based defintion of a SM scheme that could be lifted and, second, we require the specific simulation technique encoded in most game-based definitions, in contrast to the existential simulator of our constructive SM statements.

6.1 Overview

Receiver non-committing encryption. The technical tool we use to construct adaptively-secure secure-messaging (ASM) schemes with optimal security is so-called receiver non-committing encryption (RNCE), introduced by Canetti et al. [5]. Intuitively, in RNCE schemes, key generation outputs an additional trapdoor z, ignored by honest parties and used by the simulator. Then, there are two ways to generate a ciphertext: (1) an "honest" ciphertext is computed in the standard way $c \leftarrow \mathsf{RNCE.E}(pk, m)$ (so, as in any encryption scheme, it is a commitment to the message), (2) a "fake" ciphertext is computed (by the simulator) without the message, but with the secret key sk and the trapdoor z as $\tilde{c} \leftarrow \mathsf{RNCE.F}(pk, sk, z)$. Given a fake ciphertext $\tilde{c}$ and any message m, one can compute a secret key $\tilde{sk} \leftarrow \mathsf{RNCE.R}(pk, sk, z, \tilde{c}, m)$ that explains the message-ciphertext pair (such that $\mathsf{RNCE.D}(\tilde{sk}, \tilde{c}) = m$). Moreover, the distributions (c, sk) (as in the real world) and $(\tilde{c}, \tilde{sk})$ (as in the simulation) are indistinguishable. This allows to explain a single ciphertext per public key.

The scheme. At a high level, the authors of [5] use RNCE to construct non-committing forward-secure public-key encryption by encrypting with a standard forward-secure public-key scheme RNCE ciphertexts instead of messages. We generalize this idea (and the simulation technique) to SM protocols. In particular, we can construct an ASM scheme by taking a standard SM scheme that suffers from the commitment problem and sending, instead of messages, their RNCE encryptions, where each message is encrypted with a different public key. When

[5] In game-based definitions, one can think of the "corrupt" oracle not being silenced even if the challenge has been issued, but instead outputting the secret state corresponding to the challenge bit 0.

a message is received, the secret key is immediately deleted. (For the moment, assume that whenever Alice sends a message, an RNCE key pair is "magically" generated — Alice uses the public key, and the secret key immediately appears stored in Bob's state.) This way, the modified scheme inherits all guarantees of the original SM scheme. Furthermore, it can be simulated in the adaptive setting, as we will see below.

Let us now address the problem of how the RNCE keys are distributed. One trivial solution would be to include ℓ key pairs as part of the setup: the parties send their ℓ public keys at the beginning over an authenticated channel. First, this way we can send only ℓ messages overall. But even worse, the RNCE keys do not heal: when the receiver is corrupted for the first time, the simulator can explain all messages sent so far, but it also has to commit to all RNCE secret keys. Hence, adaptive security is never restored. To deal with this, we use the technique used in all SM schemes: we send with each message an update, consisting of ℓ fresh RNCE public keys. In particular, Alice (Bob will proceed analogously) stores some public keys previously received from Bob. When she sends the i-th message, she RNCE-encrypts it with one of the unused public keys, generates ℓ new key pairs, stores the secret keys, and sends the RNCE ciphertext, the ℓ public keys and i to Bob over the channel constructed by an SM scheme. Bob stores the greatest index i he has seen so far. Whenever he sees a messsage with a greater i, he ignores all RNCE public keys he has and replaces them by the ℓ newly received ones. Unlike in the first trivial solution, in the above protocol adaptive security is restored as fast as possible: with the first new message delivered from the other party.

Simulation. We give an intuition of how the above protocol can be simulated. Assume that the SM scheme has the standard simulator, as hard-coded in most game-based definitions. In particular, he executes the protocol, and when a memory is exposed, he shows to the distinguisher the real state. For ciphertexts corresponding to confidential messages it shows encryptions of 0's, while for non-confidential ones it shows encryptions of the actual message.

In the adaptive setting, the real and the ideal world are easily distinguishable for that simulator. This is because when a message is sent as confidential, and later the memory is exposed, the distinguisher sees in the ideal world the encryption of 0's. However, we can fix this with our new scheme: the new simulator encrypts, instead of 0's, a fake RNCE ciphertext to generate a ciphertext corresponding to a confidential message. When a memory is corrupted, he receives the message (which, of course, can no longer be confidential) and computes the fake RNCE secret key according to the fake ciphertext. RNCE guarantees that this is indistinguishable from the real world, where we have honest ciphertext and an honest key.

A note on efficiency. First, observe that using a symmetric non-committing encryption scheme, such as the one-time pad, instead of RNCE would not work. This is because in many SM schemes corrupting the sender has no effect on confidentiality, implying that upon such a corruption, the simulator needs to

output a key of the symmetric non-committing scheme without knowing the messages (which trivially breaks against a distinguisher knowing the message).

Moreover, while our construction of using nested encryption appears to be redundant, it can be observed that using RNCE only would not suffice. This is because SM schemes can provide certain advanced *confidentiality* guarantees not achieved by RNCE alone. For example, the optimal schemes such as [9, 23] provide so-called post-impersonation guarantees: once the adversary injects a message to Bob (after corrupting Alice) and then corrupts Bob, all messages sent by Alice afterwards are confidential.

Limitations. Our protocol requires a fixed upper bound on the number of messages a party can send without interaction (in particular, after ℓ messages it needs a new set of public keys from the partner). Unfortunately, overcoming this seems unlikely with our approach. This is due to the impossibility result by Nielsen [19]. It essentially says that a non-committing non-interactive public-key encryption scheme requires that the length of a secret key is at least the overall length of all messages encrypted. This means that we would need non-committing encryption, where the public and secret keys are updated, in other words, a non-committing equivalent of HIBE. To the best of our knowledge, this does not exist yet.[6]

6.2 The Construction: Combining RNCE with HIBE

Recall that the HIBE protocol from Section 5.2 is designed for the sesqui-directional setting, where it protects the confidentiality of messages sent by Alice. In the protocol, Bob sends to Alice HIBE master public keys, which results in epochs. In epoch j, Alice uses the j-th master public key to encrypt her messages with the transcript as identity. In this section we consider the analogous setting for the ASM protocol, consisting of RNCE composed with HIBE. That is, Bob sends ℓ RNCE keys alongside the HIBE keys, and Alice uses them to additionally encrypt her messages.

Hence, for the ASM construction we need in the real world the additional randomness $\mathsf{Rand}^{renc_i,\mathtt{A}}$ for RNCE-encrypting the i-th message and $\mathsf{Rand}^{rkg_j,\mathtt{B}}$ for generating the j-th set of ℓ keys, compared to the real world from the HIBE protocol. Moreover, we have memories $\mathsf{Mem}^{rsk_{(j,k)},\mathtt{B}}$ for storing the k-th RNCE secret key, generated in epoch j, and insecure (rewritable) memories $\mathsf{IMem}^{rpk,\mathtt{A}}$ for storing the set of RNCE public keys. Overall, the real-world resources are as follows.

$$\mathsf{R}^{\textsf{ad-hibe}} := \Big[\mathsf{R}^{\textsf{hibe}}, \big\{\mathsf{Rand}^{renc_i,\mathtt{A}}\big\}_{i\in[n]}, \big\{\mathsf{Rand}^{rkg_j,\mathtt{B}}\big\}_{j\in[n]}, \mathsf{IMem}^{rpk,\mathtt{A}}, \big\{\mathsf{Mem}^{rsk_{(j,k)},\mathtt{B}}\big\}_{j\in[n],k\in[\ell]}\Big], \quad (10)$$

[6] Note that the impossibility of [19] also rules out a solution where Alice RNCE-encrypts for Bob a new RNCE secret key, used for the next message — this secret key would leave no space for the message.

where $\mathsf{R}^{\mathsf{hibe}}$ should be understood as the same set of resources as in Section 5.2. The restrictions on those set of resources are dropped, on the other hand, since we no longer need work around the commitment problem. This implies, however, that we have to directly consider security of the overall compiled protocol, instead of using the construction statement for HIBE and composition.[7] A formal description of the converters $\mathsf{rnce\text{-}enc}$ and $\mathsf{rnce\text{-}dec}$ implementing the RNCE protocol on top of the HIBE protocol is given in Figure 8.

In the ideal world, we have the same $2n$ channels: $\mathsf{S}^{\mathsf{ad\text{-}hibe}} := \mathsf{S}^{\mathsf{hibe}}$. Most properties of the constructed channels are the same as in the HIBE construction. In fact, our adaptive protocol only affects (1) availability — only ℓ messages can be sent without interaction, and (2) confidentiality — we need to account for the additional randomness and memory resources. Recall that the epoch j_i in which message i is sent by Alice is determined by the sent and received events. With this, the restriction (1) can be expressed with the can-send and can-receive predicate in a straightforward way.

Let us now focus on confidentiality. Recall that in the HIBE protocol, the can-leak predicate was defined using the event $\mathcal{E}_i^{\mathsf{exposed}}$, denoting that the i-th message sent by Alice is inherently insecure. We modify this event to account for the additional resources used by RNCE. Specifically, the message is exposed if the RNCE-encryption randomness leaks: $\mathcal{E}^{\mathsf{leaked}}_{\mathsf{Rnd}(renc_i,\mathtt{A})}$, or if the RNCE secret key leaks. The latter happens if Bob's key-generation randomness leaks: $\mathcal{E}^{\mathsf{leaked}}_{\mathsf{Rnd}(rkg_{j_i},\mathtt{B})}$, or if the secret key memory leaks: $\mathcal{E}^{\mathsf{leaked}}_{\mathsf{Mem}(rsk_{(j_i,k_i)},\mathtt{B})}$, where the i-th message was the k_i-th one sent in its epoch. Overall, this leads to the following composed event:

$$\mathcal{E}_i^{\mathsf{exposed\text{-}ad}} := \mathcal{E}_i^{\mathsf{exposed}} \vee \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Rnd}(renc_i,\mathtt{A})} \vee \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Rnd}(rkg_{j_i},\mathtt{B})} \vee \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Mem}(rsk_{(j_i,k_i)},\mathtt{B})}.$$

The leakage function $\mathfrak{L}^{\mathsf{S}^{\mathsf{ad\text{-}hibe}}}_{\mathsf{Ch}(\mathtt{A}\to\mathtt{B})}$ is then defined analogously to that of the HIBE construction `silent` in case of $\mathcal{E}_i^{\mathsf{exposed\text{-}ad}}$, and `false` otherwise.

We stress that the need to include these additional cases only arises from our fine-grained modeling of memory and randomness. In reality, it makes sense to consider only one memory storing the whole secret state, only one randomness for RNCE and HIBE encryption, and so on. In such a model, the confidentiality of our adaptively secure scheme and the non-adaptive one would coincide.

The security of our composed protocol is summarized in the following theorem.

Theorem 3. *Let* $\mathsf{R}^{\mathsf{ad\text{-}hibe}}$ *be as in* (10)*, and let* $\mathsf{S}^{\mathsf{ad\text{-}hibe}}$ *be as in above with the described confidentiality guarantees, in-order sending and receiving, and the restriction to* ℓ *messages per epoch. If the HIBE scheme is IND-CCA secure with our additional assumptions, then there exists an efficient simulator* sim*, such that*

$$\mathsf{rnce\text{-}enc}^{\mathtt{A}}\mathsf{hibe\text{-}enc}^{\mathtt{A}}\ \mathsf{rnce\text{-}dec}^{\mathtt{B}}\mathsf{hibe\text{-}dec}^{\mathtt{B}}\ \mathsf{R}^{\mathsf{ad\text{-}hibe}} \approx_\tau \mathsf{sim}^{\mathtt{E}}\ \mathsf{S}^{\mathsf{ad\text{-}hibe}},$$

[7] In general, the simulator for the SM scheme simply does not output the secret state from the commitment-causing memories, and our ASM simulator cannot generate it himself, since this would be inconsistent with the rest of the SM simulation.

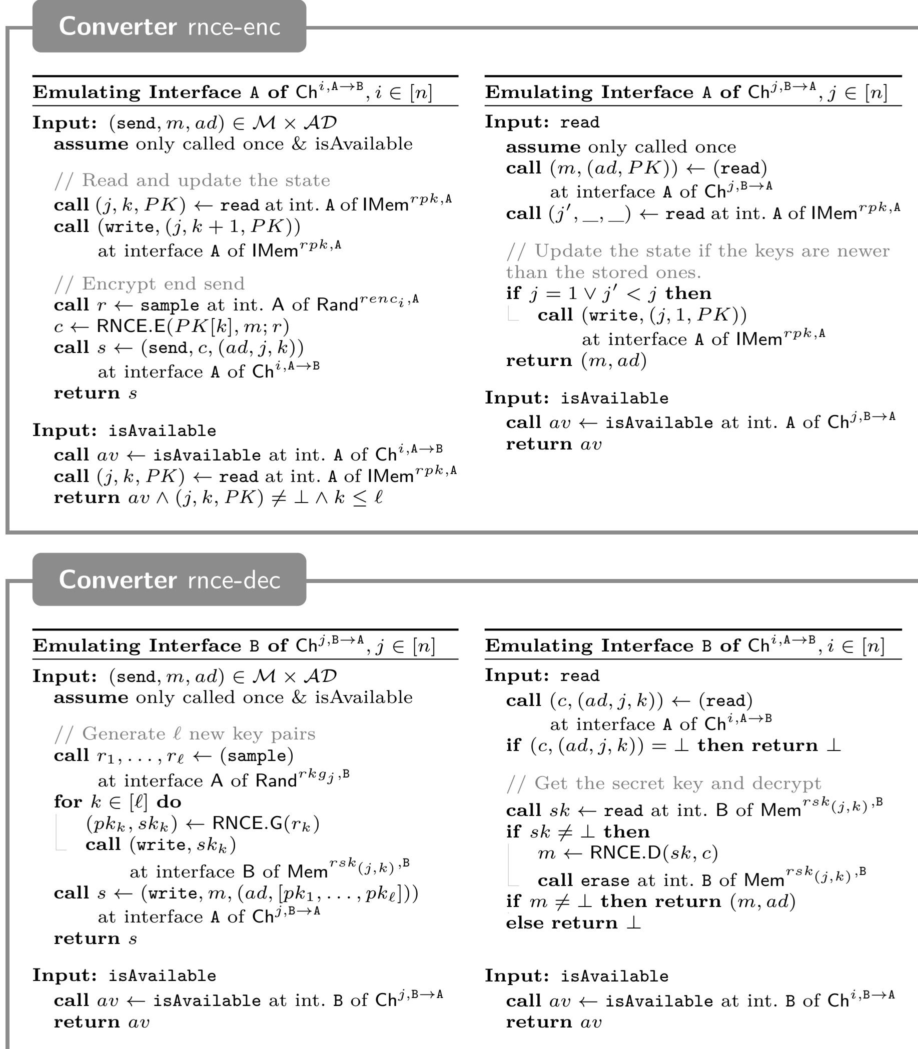

Fig. 8: The RNCE part of the adaptively-secure protocol in the sesqui-directional setting.

where τ is the same event mapping as in Theorem 2.

Proof (sketch). We sketch the simulator and simultaneously also argue why this simulation strategy makes the two worlds actually indistinguishable. Note that we focus here only on the RNCE parts, referring to the proof of Theorem 2, as to why the HIBE scheme provides proper healing, forward secrecy, and post-impersonation security.

The simulator essentially executes $\mathsf{sim}_{\mathsf{hibe}}$ (cf. Appendix B.2), except that it also internally samples all RNCE keys, including the trapdoors, and instead of encrypting a zero-string, it encrypts a fake RNCE ciphertext.

The randomness and public-key memory for RNCE are trivially simulatable. Furthermore, all memory and randomness resources of the HIBE construction are simulated as in $\mathsf{sim}_{\mathsf{hibe}}$. Clearly, we can also simulate both the `read` and `deliver` for the channels from Bob to Alice, where for instance *same* is determined analogous to $\mathsf{sim}_{\mathsf{hibe}}$ just also taking those public keys into account.

More interesting is the simulation of the channels from Alice to Bob (whose confidentiality the scheme protects) and the memories storing the corresponding RNCE secret keys. We consider three cases: (1) the parties are in sync (before an active impersonation) and the adversary simply forwards the message, (2) the parties are in sync and adversary tries to inject a message, and (3) the parties are already out of sync.

In sync. Consider one channel and the associated secret key, where the parties are in sync and the adversary does not carry out a successful impersonation. On every `read` input to the channel, the simulator outputs either a proper encryption (if the message is known), or a HIBE-encryption of a fake ciphertext. For the key leakage, there are the following options:

- The distinguisher did not input the message to be encrypted under that key yet. The simulator outputs the honest secret key. Later on the `read` input to the channel, the message will be known to the simulator due to our can-leak predicate. Hence, the simulation is perfect.
- The message has been received. The simulator outputs $\perp$, since in the real world Bob would erase the key.
- A message is in transmission. Due to our can-leak predicate, this message is revealed to the adversary, so the simulator can produce a fake secret key that explains it. Indistinguishability follows from the security of RNCE.

For the `deliver` command, we proceed the same way as $\mathsf{sim}_{\mathsf{hibe}}$: we either forward the message (if it is the same ciphertext), or trigger an error (by our additional assumption on the HIBE scheme).

First injection. Consider now a channel, where parties are still in sync, and the adversary tries to deliver her own message. We proceed the same way as $\mathsf{sim}_{\mathsf{hibe}}$ and just decrypt the ciphertext and observe the result, where the following RNCE is used:

- If the simulator already output a fake RCNE secret key, then it uses that one to decrypt.

– Otherwise, it uses the *honest* RNCE secret key. Here security follows from the CCA1-security of RNCE: a fake key and ciphertext are indistinguishable from the honest ones even given decryptions of adversarially chosen messages under the honest key before the fake key has been produced. Hence, even if the simulator later has to provide a fake RNCE key that explains Alice's message, this is indistinguishable from the real world.

Out of sync. Once the parties are out of sync, the adversary is allowed to learn the RNCE-keys without revealing the messages to the simulator (and thus allowing the simulator to output a fake secret key). The confidentiality of all channels after an impersonation attack is, however, guaranteed by the HIBE protocol, even if Bob's state is fully revealed to the adversary. Thus, the adversary never actually gets to see the fake RNCE ciphertexts (which are encrypted with the HIBE-scheme). As a consequence, the real RNCE keys can be safely revealed. □

7 Asynchronous Ratcheting as Continuous Key Agreement

Many secure messaging protocols, including Signal, proceed by combining some form of continuous key agreement (the asynchronous ratcheting layer) with a forward-secure symmetric encryption scheme (the synchronous ratcheting layer). Thus, the continuous key agreement primitive appears to be a natural abstraction boundary. Indeed, Alwen et al. [1] modularize and abstract Signal in this manner by formalizing a notion of Continuous Key Agreement (CKA), which they then combine (with the help of a special PRF) with Forward-Secure AEAD (FS-AEAD).

In this section, we outline how such a CKA notion can be naturally expressed within our framework as a protocol that uses a given communication network to construct a sequence of keys (while preserving the communication network). We thereby focus on CKA only—the use of FS-AEAD to achieve secure messaging would then follow along the same lines as in [1].

Continuous key agreement. We first briefly recall the CKA primitive, and refer to [1] for further details. The setting is that of interlocked (or 'ping-pong') communication over authenticated channels, initiated by Alice. This means that in each odd round (which, following [1], we call an *epoch*), Alice sends a message to Bob, and in each even round Bob sends a message to Alice. The goal is to provide a continuous stream of keys: each sending and receiving operation outputs a symmetric key (later used in the symmetric ratchet). This means that in each epoch (when a single message is sent and received) the parties produce a new key in the key stream (see Figure 9). Formally, a CKA scheme consists of four algorithms, of which $\mathsf{CKA.S}$ is randomized and the others deterministic:

$\mathsf{CKA.I\text{-}A}$ ($\mathsf{CKA.I\text{-}B}$): On input a shared secret k, output the initial state γ^{A} of Alice (the initiator) (γ^{B} of Bob (the responder)).

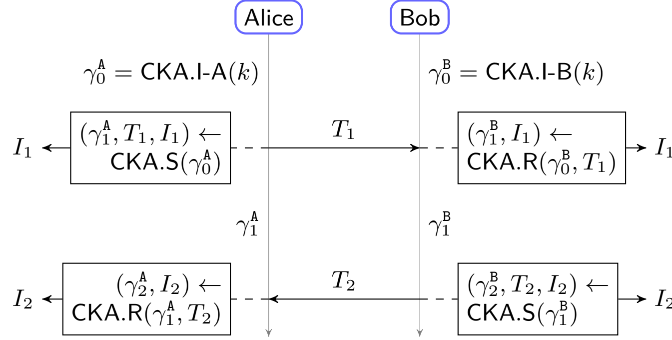

Fig. 9: The synchronous execution of CKA, epochs 1 and 2. Alice sends in odd epochs, while Bob sends in even ones. In epoch 1, the parties produce the key I_1, which is available to Alice (the initiator in this key agreement) immediately, and to Bob (the responder) only after he receives the message T_1.

CKA.S: On input a sending state $\gamma^{\mathtt{A}}$, output a new receiving state, an update message T (sent to the partner), and the next shared key I.

CKA.R: On input a receiving state $\gamma^{\mathtt{B}}$ and an update message T (received from the partner), output a new sending state and the next shared key I.

For correctness, we want that both parties produce the same key stream. The standard security properties include indisitnguishability: each key is indistinguishable from an (independent) random one, even given a number of other keys. Moreover, in case of a secret-state compromise, we require (1) forward secrecy: security of previously generated keys is not affected, and (2) healing: after a number of epochs since the last compromise, the security is restored. The scheme is parameterized by Δ, denoting the number of epochs that need to pass since epoch e until the state contains no information about the e-th key. We refer to [1] for a formal description of the CKA security game.

The constructed key stream. In contrast of the previously presented constructions, the goal of a CKA protocol is not to enhance the guarantees of the channels (at least not directly). Rather, the goal is to construct a sequence of keys while preserving the channels and their respective guarantees.

We model this sequence of keys as the parallel composition of many individual key resources $\mathsf{Key}^{e,(\mathtt{A},\mathtt{B})}$ shared between Alice and Bob, where e denotes the epoch number. A formal definition of the key resource is given in Figure 10. On a high level, it essentially behaves like a pair of memory resources that have the respective key stored. More concretely, the key's interfaces and their capabilities are as follows:

- Parties can read the key once it is available to them. The availability is determined via the respective can-read predicates $\mathfrak{R}_{\mathsf{I}}$ and $\mathfrak{R}_{\mathsf{R}}$. Moreover, the parties can each request to securely erase their respective copy of the key.

- The adversary E can potentially leak the key from either party, if this is allowed by the respective can-leak predicate $\mathfrak{L}_{\mathtt{U}}$, and the key has not been securely erased yet. Note that the leaked key still looks random.
- Moreover, in certain circumstances the key is no longer random (e.g., if the secret state used to produce it leaks). This corresponds to the adversary being able to inject her own key. Specifically, the adversary can inject, for each party, a function *inj*, that is invoked when a party fetches their key and the can-inject predicate evaluates to true. The function computes the injected key, given the current event history.

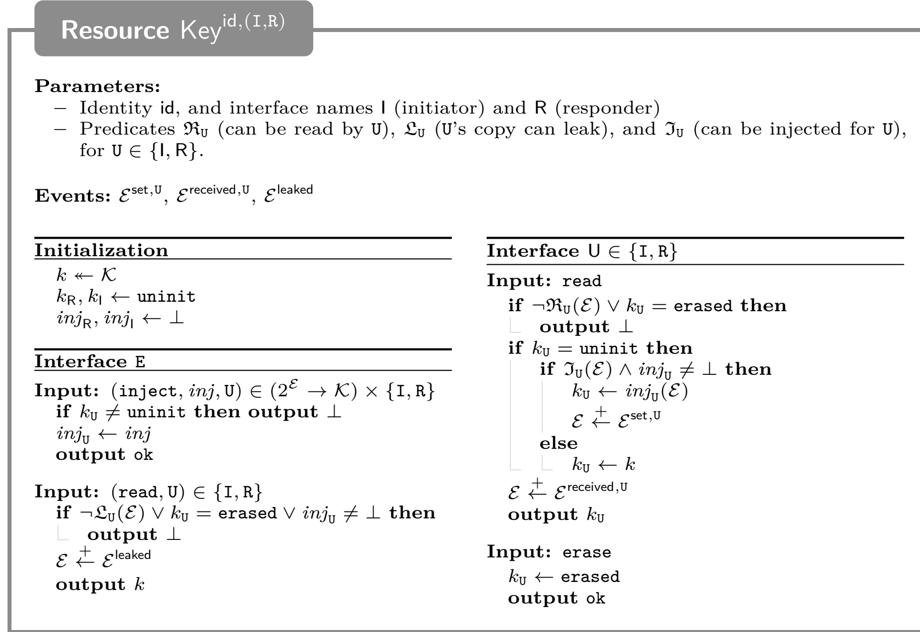

Fig. 10: A formal description of the key resource. Note that the adversary can not only learn the key, but (if allowed) also set it individually for both parties. Note that for setting the key he is allowed to submit a function *inj* that determines the key (upon when the party first fetches it) based on the event history. That is, the party obtains $inj(\mathcal{E})$.

The setting. Although CKA is a primitive that mandates an interlocked communication pattern, it is still intended to be used on top of asynchronous communication between Alice and Bob. In particular, a party will attach the latest synchronous update message T to each asynchronous message it sends. This way, the CKA protocol proceeds as fast as possible: as soon as at least

one of the messages carrying a new, previously unseen update arrives, a party can move to the next epoch. For this reason, the real world involves the usual sequence of (insecure) channels between Alice and Bob:

$$\mathsf{Channels} := \Big[\big\{ \mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}} \big\}_{i\in[n]}, \big\{ \mathsf{Ch}^{j,\mathtt{B}\to\mathtt{A}} \big\}_{j\in[n]} \Big].$$

In the real world, we have various memory resources: Each user $\mathtt{U}$ stores his current epoch number e in the insecure memory $\mathsf{IMem}^{ep,\mathtt{U}}$, and he stores his CKA state corresponding to epoch e in $\mathsf{Mem}^{st_e,\mathtt{U}}$. Moreover, he stores the key I_e produced in epoch e in $\mathsf{OMem}^{key_e,\mathtt{U}}$ (where OMem denotes an observable memory that behaves like Mem, except that it triggers an $\mathcal{E}^{\mathsf{read}}$ event when the honest reader accesses the content for the first time[8]). Overall, we have

$$\mathsf{Mems} := \Big[\big\{ \mathsf{Mem}^{st_e,\mathtt{U}}, \mathsf{OMem}^{key_e,\mathtt{U}}, \big\}_{e\in[2n],\mathtt{U}\in\{\mathtt{A},\mathtt{B}\}}, \big\{ \mathsf{IMem}^{ep,\mathtt{U}} \big\}_{\mathtt{U}\in\{\mathtt{A},\mathtt{B}\}} \Big],$$

Furthermore, there are randomness resources $\mathsf{Rand}^{e,\mathtt{U}}$ for the $\mathsf{CKA.S}$ operation:

$$\mathsf{Rand} := \Big[\big\{ \mathsf{Rand}^{e,\mathtt{A}} \big\}_{odd\ e\in[2n]}, \big\{ \mathsf{Rand}^{e,\mathtt{B}} \big\}_{even\ e\in[2n]} \Big].$$

Finally, CKA requires setup in the form of a shared key, which is used by Alice and Bob to derive their initial states via $\mathsf{CKA.I\text{-}A}$ and $\mathsf{CKA.I\text{-}B}$, respectively. Security of this operation is, however, outside the scope of CKA (in particular, the shared key is never revealed to the adversary). We model this by a real-world setup resource $\mathsf{CKA\text{-}Setup}^{(\mathtt{I},\mathtt{R})}$, formally defined in Figure 11, that executes the initialization procedure and provides the initial states $\gamma_0^{\mathtt{A}}$ and $\gamma_0^{\mathtt{B}}$ to the respective parties. That is, analogous to our key-resource, the setup resource essentially corresponds to a pair of memories $[\mathsf{Mem}^{st_0,\mathtt{A}}, \mathsf{Mem}^{st_0,\mathtt{B}}]$, except that instead of the parties writing to it, their states are "magically" initialized.

Putting it all together, we have the following assumed resources:

$$\mathsf{R}_{\mathsf{CKA}} := \Big[\mathsf{CKA\text{-}Setup}^{(\mathtt{A},\mathtt{B})}, \mathsf{Channels}, \mathsf{Mems}, \mathsf{Rand} \Big], \tag{11}$$

to which the converters implementing the CKA protocol will be attached.

The protocol. As we already hinted before, the protocol works by attaching to each message sent on one of the channels the next CKA message T. That is, we execute the protocol sketched on Figure 9, except each message T_e is repeated with each asynchronous message.

Figure 12 formally describes the protocol executed by Alice. Her state in epoch e is stored in $\mathsf{Mem}^{st_e,\mathtt{A}}$. For an even e, this is a 'receiving' state (recall that Alice can only receive messages in such epochs), that contains simply the CKA state γ. For an odd e, this is a 'sending' state, containing the pair (γ, T), where T

[8] We need this to prevent the commitment problem in case an ideal key has been output.

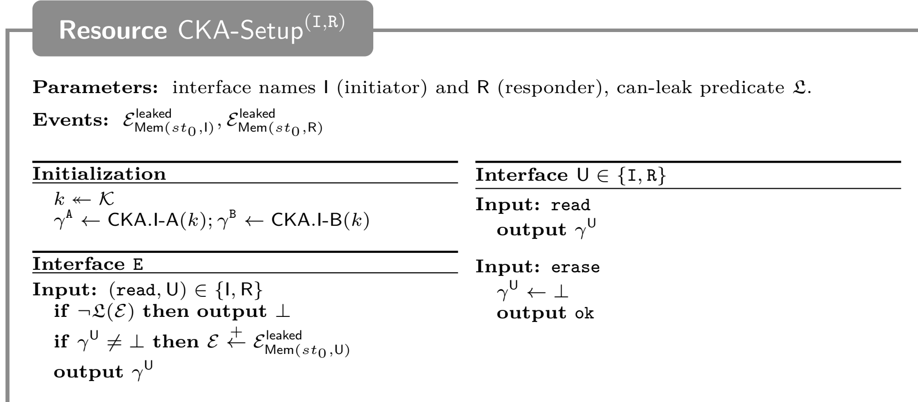

Fig. 11: The resource encoding the setup and initialization of CKA.

is the current update message, attached to all Alice's messages in this epoch. On each `write` input on the outside interface of a channel, the protocol first decides whether this message initiates a new epoch (if e is even) or not (if e is odd). If the protocol stays in the same epoch, then it fetches the 'sending' state (γ, T) from $\mathsf{Mem}^{st_e,\mathtt{A}}$ and simply attaches T to the message. Otherwise, the protocol uses the 'receiving' state fetched from the memory for the previous (even) epoch to compute the update T. This latter operation produces, as a byproduct, a new key I, which is stored in $\mathsf{OMem}^{key_e,\mathtt{A}}$. A `read` input is processed analogously. First, the message and the update T are read from the real channel. If T has not been seen before, it is used to initialize $\mathsf{Mem}^{st_e,\mathtt{A}}$, for the new epoch e, with a new receiving state. On input `read` on the outside interface of a key, the key is retrieved from $\mathsf{Mem}^{key_e,\mathtt{A}}$.

Bob's protocol essentially works analogously—swapping odd and even epochs. There are some minor differences with respect to initialization that reflect the fact that overall Alice acts as initiator and Bob as responder, i.e., Bob cannot send a message before having received one. We omit a formal specification of the respective converter cka-responder, since the necessary modifications should be self explanatory.

The construction. As mentioned before, the goal of the CKA protocol is to construct a sequence of keys while preserving the channels and their respective guarantees. Thus, the ideal world consist of channels and the (potentially) constructed key resources:

$$\mathsf{S}_{\mathsf{CKA}} \coloneqq \left[\left\{\mathsf{Key}^{e,(\mathtt{A},\mathtt{B})}\right\}_{e\in[2n]}, \mathsf{Channels}\right], \tag{12}$$

where $2n$ is an upper bound of the number of epochs that can be initiated when the parties send at most n messages each. If fewer epochs occur, then we model

Converter cka-initiator

Emulating Interface A of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$, $i \in [n]$

Input: $(\mathtt{send}, m, ad)$
- **assume** only called once & isAvailable
- **call** $e \leftarrow \mathtt{read}$ at int. A of $\mathsf{IMem}^{ep,\mathtt{A}}$
- **if** $e = \bot \lor e \bmod 2 = 0$ **then**
  - **if** $e = \bot$ **then**
    - **call** $\gamma \leftarrow (\mathtt{read})$ at interface A of $\mathsf{CKA\text{-}Setup}^{(\mathtt{A},\mathtt{B})}$
    - **call** $\mathtt{erase}$ at int. A of $\mathsf{CKA\text{-}Setup}^{(\mathtt{A},\mathtt{B})}$
    - $e \leftarrow 1$
  - **else**
    - **call** $\gamma \leftarrow \mathtt{read}$ at int. A of $\mathsf{Mem}^{st_e,\mathtt{A}}$
    - **call** $\mathtt{erase}$ at int. A of $\mathsf{Mem}^{st_e,\mathtt{A}}$
    - $e \leftarrow e+1$
  - **call** $r \leftarrow \mathtt{sample}$ at int. A of $\mathsf{Rand}^{e,\mathtt{A}}$
  - $(\gamma, T, I) \leftarrow \mathsf{CKA.S}(\gamma; r)$
  - **call** $(\mathtt{write}, e)$ at int. A of $\mathsf{IMem}^{ep,\mathtt{A}}$
  - **call** $(\mathtt{write}, \gamma, T)$ at int. A of $\mathsf{Mem}^{st_e,\mathtt{A}}$
  - **call** $(\mathtt{write}, I)$ at int. A of $\mathsf{OMem}^{key_e,\mathtt{A}}$
- **else**
  - **call** $(\gamma, T) \leftarrow \mathtt{read}$ at int. A of $\mathsf{Mem}^{st_e,\mathtt{A}}$
- **call** $succ \leftarrow (\mathtt{send}, m, (ad, e, T))$ at interface A of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
- **return** $succ$

Input: $\mathtt{isAvailable}$
- **call** $succ \leftarrow \mathtt{isAvailable}$ at int. A of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
- **return** $succ$

Emulating Interface A of $\mathsf{Ch}^{j,\mathtt{B}\to\mathtt{A}}$, $j \in [n]$

Input: $\mathtt{receive}$
- **assume** only called once & isAvailable
- **call** $e \leftarrow \mathtt{read}$ at int. A of $\mathsf{IMem}^{ep,\mathtt{A}}$
- **call** $(m, (ad, e', T')) \leftarrow (receive)$ at interface A of $\mathsf{Ch}^{j,\mathtt{B}\to\mathtt{A}}$
- **if** $\neg(2 \le e' \le e+1) \lor e' \bmod 2 = 1$ **then**
  - **return** $\bot$
- **else if** $e' = e+1$ **then**
  - **call** $(\gamma, T) \leftarrow \mathtt{read}$ at int. A of $\mathsf{Mem}^{st_e,\mathtt{A}}$
  - $(\gamma, I) \leftarrow \mathsf{CKA.R}(\gamma, T')$
  - $e \leftarrow e+1$
  - **call** $(\mathtt{write}, e)$ at int. A of $\mathsf{IMem}^{ep,\mathtt{A}}$
  - **call** $\mathtt{erase}$ at int. A of $\mathsf{Mem}^{st_{e-1},\mathtt{A}}$
  - **call** $(\mathtt{write}, \gamma)$ at int. A of $\mathsf{Mem}^{st_e,\mathtt{A}}$
  - **call** $(\mathtt{write}, I)$ at int. A of $\mathsf{OMem}^{key_e,\mathtt{A}}$
- **return** (m, ad)

Input: $\mathtt{isAvailable}$
- **call** $e \leftarrow \mathtt{read}$ at int. A of $\mathsf{IMem}^{ep,\mathtt{A}}$
- **call** $succ \leftarrow \mathtt{isAvailable}$ at int. A of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
- **return** $e \neq \bot \land succ$

Emulating Interface A of $\mathsf{Key}^{e,(\mathtt{A},\mathtt{B})}$, $e \in [2n]$

Input: $\mathtt{read}$
- **call** $k \leftarrow \mathtt{read}$ at int. A of $\mathsf{OMem}^{key_e,\mathtt{A}}$
- **return** k

Input: $\mathtt{erase}$
- **call** $\mathtt{erase}$ at int. A of $\mathsf{OMem}^{key_e,\mathtt{A}}$
- **return** $\mathtt{ok}$

Fig. 12: The protocol cka-initiator implementing CKA, executed by the initiator, Alice. The protocol cka-responder executed by the responder, Bob, is analogous.

this by simply *not enabling* the corresponding key resources, i.e, the resources formally exist but are not available to the parties.

Let us now describe the properties of the constructed key sequence, as determined by the predicates can-read, can-leak, and can-set. Both parties can read the key $\mathsf{Key}^{e,(\mathtt{A},\mathtt{B})}$ as soon as they entered the e-th epoch:

$$\mathfrak{R}^{\mathsf{S_{CKA}}}_{\mathtt{U},\mathsf{Key}(e,(A,B))}(\mathcal{E}) :\Leftarrow \mathcal{E}^{\text{in-epoch}}_{e,\mathtt{U}}, \tag{13}$$

where $\mathcal{E}^{\text{in-epoch}}_{e,\mathtt{U}}$ denotes the event that user U entered the e-th epoch. Note that as long as Eve does not inject a forgery, this can easily be determined from the event history that contains the sequence of all sent and received messages and their respective order. Once Eve injects, we can, for the sake of those predicates, assume that she will always advance the session (cf. the simulator in the proof). Hence, the event $\mathcal{E}^{\text{in-epoch}}_{e,\mathtt{U}}$ can easily be made formal.

The can-leak predicate of a key corresponds to the can-leak predicate of the memory storing it, that is it leaks to the adversary if the memory leaks:

$$\mathfrak{L}^{\mathsf{S_{CKA}}}_{\mathtt{U},\mathsf{Key}(e,(A,B))}(\mathcal{E}) := \mathfrak{L}^{\mathsf{R_{CKA}}}_{\mathtt{U},\mathsf{Mem}(key_e,\mathtt{U})}(\tilde{\mathcal{E}}), \tag{14}$$

which captures the situations where key directly leaks.

The stronger corruption, where Eve sets the key, is controlled by the can-set predicate. This models situations, in which a key is no longer random (and hence can be set by Eve). Roughly, keys are not random in two situations: First, a key for epoch e is not random if either the randomness to generate it leaked, or a user's state leaks in any epoch $e' \in \{e-1, \ldots, e+\Delta-1\}$. Note that this is just a safe approximation. For instance, while the scheme could heal for $\mathtt{U}$ between epoch $e-1$ and e (if e is a sending epoch for him) or not (otherwise), it is guaranteed to heal between epoch $e-2$ and e, and analogous for $e+\Delta$. Since the can-set predicate is only defined on past events, for now we only say that it is true if a user's state leaked in epochs $e-1$ or e. The other epochs $e+1, \ldots, e+\Delta-1$ are considered when we deal with the commitment problem—we will assume that the state in these epochs does not leak. As a consequence, we define the following composed event:

$$\mathcal{E}^{\text{state-leaked}}_{\mathtt{U},e} := \mathcal{E}^{\text{leaked}}_{\mathsf{Mem}(st_e,\mathtt{U})} \vee \mathcal{E}^{\text{leaked}}_{\mathsf{Mem}(st_{e-1},\mathtt{U})} \vee \mathcal{E}^{\text{leaked}}_{\mathsf{Rand}(e,\mathtt{U})}.$$

Second, all keys for epochs following an active attack are not random either (we give up on any guarantees in such case; originally, CKA does not consider injections at all). Specifically, assume that the adversary injects the i-th message. Clearly, she can influence the receiver's key of the corresponding epoch e_i. In addition, we allow the adversary to set the keys of all upcoming epochs e, and thus give up all guarantees for epochs e with $e_i \leq e$, as expressed by the following event:

$$\mathcal{E}^{\text{injected-before}}_{\mathtt{U}\to\bar{\mathtt{U}},e} := \exists i : \exists e_i \leq e : \left(\mathcal{E}^{\text{in-epoch}}_{e_i-1,\mathtt{U}} \prec \mathcal{E}^{\text{sent}}_{\mathsf{Ch}(i,\mathtt{U}\to\bar{\mathtt{U}})} \prec \mathcal{E}^{\text{in-epoch}}_{e_i+1,\mathtt{U}}\right) \wedge \mathcal{E}^{\text{injected}}_{\mathsf{Ch}(i,\mathtt{U}\to\bar{\mathtt{U}})},$$

where the first condition just asserts that e_i is indeed the i-th message's epoch.

Finally, for the epoch of the first injection, CKA does not guarantee the randomness of neither the receiver's nor the sender's key (by injecting the modified sender's message, Eve can cause the keys to be correlated)

Overall, we define the following can-inject predicates, which are fully symmetric for both parties:

$$\mathfrak{I}^{\mathsf{S_{CKA}}}_{\mathtt{U},\mathsf{Key}(e,(A,B))}(\mathcal{E}) := \mathcal{E}^{\text{key-exposed}}_{\mathtt{A},e} \vee \mathcal{E}^{\text{key-exposed}}_{\mathtt{B},e}, \tag{15}$$

where

$$\mathcal{E}^{\text{key-exposed}}_{\mathtt{U},e} := \mathcal{E}^{\text{state-leaked}}_{\mathtt{U},e} \vee \mathcal{E}^{\text{injected-before}}_{\bar{\mathtt{U}}\to\mathtt{U},e}.$$

The commitment problem. The ideal world $\mathsf{S_{CKA}}$ with the predicates described above is trivially distinguishable from the real world. This is because,

as already mentioned, leaking a state in epoch $e_l \in \{e+1,\dots,e+\Delta-1\}$ compromises the secrecy of the key from a past epoch e. However, in the adaptive setting, the can-set predicate for this key cannot take into account the future event of memory leakage.

In order to not have to retreat to a fully static corruption model, we deal with this in our usual way of making an assumption on the real world. That is, analogous to the game-based definition of CKA, if the distinguisher decides to "see" the ideal key in epoch e, then the memory cannot leak in epochs $e+1,\dots,e+\Delta-1$. Overall, we require

$$\mathfrak{L}^{\mathsf{R}_{\mathsf{CKA}}}_{\mathsf{Mem}(st_e,\mathtt{U})}(\mathcal{E}) \;:=\; \neg\exists \mathtt{U}', e' : \big(e-\Delta < e' \leq e\big) \wedge \big(\mathcal{E}^{\mathsf{committed}}_{\mathtt{U}',e'} \prec \mathcal{E}^{\mathsf{key\text{-}exposed}}_{\mathtt{U}',e'}\big), \quad (16)$$

where

$$\mathcal{E}^{\mathsf{committed}}_{\mathtt{U},e} := \mathcal{E}^{\mathsf{leaked}}_{\mathsf{OMem}(key_e,\mathtt{U})} \vee \mathcal{E}^{\mathsf{read}}_{\mathsf{OMem}(key_e,\mathtt{U})}$$

simply denotes the event that the key has either been output to the user or leaked to the adversary.

Moreover, the commitment problem also occurs for active injections: if the adversary chooses to change the value T to T' for the first time, then this also compromises the sender's key, which at this time might already have been used. Analogous to the CKA security game of [1], we thus have to prevent injections if the sender is committed on a uniform key for which the receiver did not yet receive the message. To this end, we require that the delivery-function is of the following type:

$$\mathfrak{D}^{\mathsf{R}_{\mathsf{CKA}}}_{\mathsf{Ch}(i,\mathtt{U}\to\bar{\mathtt{U}})}(\mathcal{E}, same) := \begin{cases} err & \text{if } \begin{array}{l} \neg same \wedge \exists e : \neg\mathcal{E}^{\mathsf{in\text{-}epoch}}_{e,\bar{\mathtt{U}}} \\ \quad \wedge\, \mathcal{E}^{\mathsf{committed}}_{\mathtt{U},e} \prec \mathcal{E}^{\mathsf{key\text{-}exposed}}_{\mathtt{U},e} \end{array} \\ \text{arbitrary} & \text{else,} \end{cases} \quad (17)$$

for an appropriate error err. Note that this only disallows the very first injection, as this immediately triggers $\mathcal{E}^{\mathsf{key\text{-}exposed}}_{\mathtt{U},e_i}$ for all future epochs.

Summary and analysis. With this workaround for the commitment problem in place, the CKA scheme now achieves the described construction, as summarized in the following theorem.

Theorem 4. *If the CKA scheme* CKA *is* (Δ,ϵ)*-secure for some negligible* ϵ*, then there exists an efficient simulator* sim *such that*

$$\mathsf{cka\text{-}initiator}^{\mathtt{A}}\ \mathsf{cka\text{-}responder}^{\mathtt{B}}\ \mathsf{R}_{\mathsf{CKA}} \quad \approx_{\tau} \quad \mathsf{sim}^{\mathtt{E}}\ \mathsf{S}_{\mathsf{CKA}},$$

where the resources $\mathsf{R}_{\mathsf{CKA}}$ *and* $\mathsf{S}_{\mathsf{CKA}}$ *are as defined in equations (11) and (12), the corresponding predicates are as defined in equations (13) to (17). Moreover, each event* $\mathcal{E}^{\mathsf{received},\mathtt{U}}_{\mathsf{Key}(e,(\mathtt{A},\mathtt{B}))}$ *corresponds to* $\mathcal{E}^{\mathsf{read}}_{\mathsf{OMem}(key_e,\mathtt{U})}$*, and each event* $\mathcal{E}^{\mathsf{leaked},\mathtt{U}}_{\mathsf{Key}(e,(\mathtt{A},\mathtt{B}))}$ *corresponds to* $\mathcal{E}^{\mathsf{leaked}}_{\mathsf{OMem}(key_e,\mathtt{U})}$*.*

Simulator $\mathsf{sim}_{\mathsf{CKA}}$

Initialization

Sample random values $(r_1, \ldots, r_{2n})$
Execute the CKA scheme for $2n$ epochs (with the above randomness), and for each epoch e store the resulting values in $\gamma^{\mathtt{A}}[e], \gamma^{\mathtt{B}}[e], I[e]$ and $T[e]$.
Initialize the empty dictionary Tr, storing all values e and T delivered in authenticated data.

Emulating Interface $\mathtt{E}$ of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$, $i \in [n]$

Input: `read`
If $\mathfrak{L}^{\mathsf{R}_{\mathsf{CKA}}}_{\mathsf{Chan}(i,\mathtt{A}\to\mathtt{B})} = \mathtt{false}$, **return** $\perp$.
call $(m, ad) \leftarrow \mathtt{read}$ at int. E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
Determine the epoch e, in which the message i was sent, using the event history.
return $(m, (ad, e, T[e]))$

Input: `readLength`
call $(\ell, ad) \leftarrow \mathtt{read}$ at int. E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
Determine the epoch e, in which the message i was sent, using the event history.
return $(\ell, (ad, e, T[e]))$

Input: $(\mathtt{deliver}, m, (ad', e', T'), same')$
assume only called once
$\mathsf{Tr}[i, \mathtt{B}] \leftarrow (e', T')$
if $T[e'] \neq T'$ **then**
 Rerun CKA scheme from e' on (with original randomness) and overwrite the respective values in $\gamma^{\mathtt{A}}[e], \gamma^{\mathtt{B}}[e], I[e]$ and $T[e]$.
call $(\mathtt{inject}, inj(\mathsf{Tr}, e, \mathtt{B}, \cdot), \mathtt{B})$
 at interface E of $\mathsf{Key}^{e,(\mathtt{A},\mathtt{B})}$
Forward the command (without e' and T') to the ideal channel
return `ok`

Input: $(\mathtt{error}, err, O, m, (ad', e', T'), same')$
Forward the input (without e' and T') to the ideal channel
return `ok`.

Function $inj(\mathsf{Tr}, e', \gamma^{\mathtt{A}}, \gamma^{\mathtt{B}}, I, \mathtt{B}, \mathcal{E})$

With the help the event history $\mathcal{E}$, the transcript Tr and the values $\gamma^{\mathtt{A}}, \gamma^{\mathtt{B}}, I$, run Bob's protocol and determine the key k he outputs in epoch e' (set $k = \perp$ if this key has not been produced yet). Output k.

Emulating Interface $\mathtt{E}$ of $\mathsf{Rand}^{e,\mathtt{A}}$, odd $e \in [2n]$

Input: `triggerLeaking`
if $\neg\mathfrak{L}^{\mathsf{R}_{\mathsf{hibe}}}_{\mathsf{Rand}(kg_j,\mathtt{B})}(\tilde{\mathcal{E}})$ **then return** $\perp$
$\mathcal{E} \xleftarrow{+} \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Rand}(e,\mathtt{A})}$
return `ok`

Input: `getLeakage`
Using the event history, determine the epoch $e_{\mathtt{A}}$, such that Alice is currently in $e_{\mathtt{A}}$.
if $e_{\mathtt{A}} < e \vee \neg\mathcal{E}^{\mathsf{leaked}}_{\mathsf{Rand}(e,\mathtt{A})}$ **then**
 return $\perp$
return r_e

Emulating Interface $\mathtt{E}$ of $\mathsf{IMem}^{ep,\mathtt{A}}$

Input: `read`
Using the event history, determine the epoch $e_{\mathtt{A}}$, such that Alice is currently in $e_{\mathtt{A}}$.
return $e_{\mathtt{A}}$

Emulating Interface $\mathtt{E}$ of $\mathsf{Mem}^{st_e,\mathtt{A}}$, $e \in \{0, \ldots, 2n\}$ (including CKA-Setup)

Input: `read`
Using the event history, determine the epoch $e_{\mathtt{A}}$ Alice is currently in.
if $e \neq e_{\mathtt{A}}$ **then return** $\perp$
$\mathcal{E} \xleftarrow{+} \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Mem}(st_e,\mathtt{A})}$
if e is even **then**
 return $\gamma^{\mathtt{A}}[e]$
else
 return $(\gamma^{\mathtt{A}}[e], T[e])$

Emulating Interface $\mathtt{E}$ of $\mathsf{OMem}^{key_e,\mathtt{A}}$, $e \in [2n]$

Input: `read`
Using the event history, determine the epoch $e_{\mathtt{A}}$, such that Alice is currently in $e_{\mathtt{A}}$.
if $e > e_{\mathtt{A}}$ **then return** $\perp$
call $k \leftarrow (\mathtt{read}, \mathtt{A})$
 at interface E of $\mathsf{Key}^{e,(\mathtt{A},\mathtt{B})}$
return k

Fig. 13: The simulator for the CKA protocol, depicting the interfaces corresponding to Alice's resources. The interface for Bob's resources are analogous.

Proof. The simulator $\mathsf{sim}_{\mathsf{CKA}}$ has to emulate the state memories and randomness resources. Moreover, it has to adjust the channel interfaces (where the CKA-values (e, T) are not present in the ideal world) and map the key resources to the corresponding memory resources in the real world. Observe that the simulator

sees all values (e, T) that get delivered to either party as the adversary needs to specify them as part of the associated data. It stores them in the transcript Tr, i.e, sets $\mathsf{Tr}[i, B] \leftarrow (e, T)$ when the message on the i-th channel to Bob contains those values. Given the event history, the simulator moreover knows which messages have been delivered, and thus can always compute the epoch a party is in.

Consider first the initial "secure" execution where the adversary only forwards messages. The simulator samples all the randomness and pre-computes the real-world states, keys and update messages for all epochs upfront. With this and the knowledge of which epoch the parties are in, he can trivially simulate the insecure memories storing the epoch number, the randomness resources, the leakage of the channels (containing (e, T)), and the state memories — at least as long as no injection has happened.

Upon a key-memory corruption, he leaks the value stored in the corresponding ideal-world key resource. It remains to show that the parties' can-inject predicates permit that (1) the "ideal" key is indistinguishable from the real key I_e if the key resource does not permit programming, or (2) he can program it consistently. Moreover, we also still need to argue that an active injection is handled properly. To this end, we consider the three stages: while the parties are still in sync, the first injection, and the phase where the parties are out of sync.

In sync. Consider the key for the e-th epoch. The CKA security game ensures that this is indistinguishable from a uniform random key unless a user's state leaks in any epoch $e' \in \{e-1, \dots, e+\Delta-1\}$, or the randomness to generate it leaked. The can-inject predicate of the key resource ensures that the simulator is allowed to program the key if a user's state of an epoch $e' \in \{e-1, e\}$, or the randomness, leaked. The commitment-workaround moreover ensures that the state of the epochs $e' \in \{e+1, \dots, e+\Delta-1\}$ does not leak if the key has been exposed in any way.
Hence, it suffices that the simulator always just tried to program the ideal-world keys to the precomputed ones (which are the same as in the real-world). In particular, every time a message for epoch e (as determined by the public associated data) is delivered to Bob by the adversary (via either `fwd` or `dlv`), $\mathsf{sim}_{\mathsf{CKA}}$ updates the $inj_{\mathtt{B}}$ function of the e-th key. The function is defined as $inj_{\mathtt{B}}(\mathsf{Tr}, e, \gamma^{\mathsf{A}}, \gamma^{\mathsf{B}}, I, \mathtt{B}, \cdot)$, where the last argument is the event history, provided by the key resource at the time the key is fetched, and the other arguments contain the whole state of the simulation. Given this, the function computes the key Bob would fetch in the real world.

First injection. Consider now a channel, where parties are still in sync, and the adversary tries to deliver her own message, including her own value T' for epoch e'. First observe that if in the ideal world, the simulator is already committed on any ideal key, especially the one for epoch e', then the injection is not allowed.
In any case, the simulator just reruns the CKA protocol from e' on. This, among others, ensures that the leakage of the channels and the state memories keeps being consistent even once the parties are out of sync. Furthermore, using the newly obtained key, the simulator programs the receiver' key as

above. Note that he already programmed the sender' key consistently, which is after the injection also the one the key resource will output by the definition of the can-inject predicate.

Out of sync. Once the parties are out of sync, we give up on any guarantee. Hence, for any delivery attempt (e', T'), the simulator can rerun the protocol from epoch e' on and consistently program all the keys. ☐

References

[1] Alwen, J., Coretti, S., Dodis, Y.: The double ratchet: Security notions, proofs, and modularization for the signal protocol. In: Ishai, Y., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2019. Springer International Publishing, Berlin, Heidelberg (2019)

[2] Bellare, M., Singh, A.C., Jaeger, J., Nyayapati, M., Stepanovs, I.: Ratcheted encryption and key exchange: The security of messaging. In: Advances in Cryptology – CRYPTO 2017. pp. 619–650 (2017)

[3] Broadnax, B., Döttling, N., Hartung, G., Müller-Quade, J., Nagel, M.: Concurrently composable security with shielded super-polynomial simulators. In: Coron, J.S., Nielsen, J.B. (eds.) Advances in Cryptology – EUROCRYPT 2017. pp. 351–381. Springer International Publishing, Cham (2017)

[4] Canetti, R.: Universally Composable Security: A New Paradigm for Cryptographic Protocols. In: 42nd IEEE Symposium on Foundations of Computer Science – FOCS 2001. pp. 136–145. IEEE Computer Society (2001)

[5] Canetti, R., Halevi, S., Katz, J.: Adaptively-secure, non-interactive public-key encryption. In: Kilian, J. (ed.) TCC 2005: 2nd Theory of Cryptography Conference. Springer, Heidelberg, vol. 3378, pp. 150–168. Springer, Heidelberg (2005)

[6] Canetti, R., Krawczyk, H.: Universally composable notions of key exchange and secure channels. In: Knudsen, L.R. (ed.) Advances in Cryptology – EUROCRYPT 2002. pp. 337–351. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)

[7] Cohn-Gordon, K., Cremers, C., Dowling, B., Garratt, L., Stebila, D.: A Formal Security Analysis of the Signal Messaging Protocol. 2nd IEEE European Symposium on Security and Privacy, EuroS and P 2017 pp. 451–466 (2017)

[8] Durak, F.B., Vaudenay, S.: Bidirectional asynchronous ratcheted key agreement with linear complexity. Cryptology ePrint Archive, Report 2018/889 (2018), https://eprint.iacr.org/2018/889

[9] Jaeger, J., Stepanovs, I.: Optimal Channel Security Against Fine-Grained State Compromise: The Safety of Messaging. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018. pp. 33–62. Springer (2018)

[10] Jost, D., Maurer, U., Marta, M.: Efficient ratcheting: Almost-optimal guarantees for secure messaging. In: Ishai, Y., Rijmen, V. (eds.) Advances in Cryptology – EUROCRYPT 2019. Springer International Publishing, Berlin, Heidelberg (2019)

[11] Kohlweiss, M., Maurer, U., Onete, C., Tackmann, B., Venturi, D.: (de-)constructing tls 1.3. In: Biryukov, A., Goyal, V. (eds.) Progress in Cryptology – INDOCRYPT 2015. pp. 85–102. Springer International Publishing, Cham (2015)

[12] Krawczyk, H.: The order of encryption and authentication for protecting communications (or: How secure is ssl?). In: Kilian, J. (ed.) Advances in Cryptology — CRYPTO 2001. pp. 310–331. Springer Berlin Heidelberg, Berlin, Heidelberg (2001)

[13] Kuesters, R., Tuengerthal, M., Rausch, D.: The iitm model: a simple and expressive model for universal composability. Cryptology ePrint Archive, Report 2013/025 (2013), https://eprint.iacr.org/2013/025

[14] Maurer, U.: Constructive Cryptography–A New Paradigm for Security Definitions and Proofs. In: Theory of Security and Applications – TOSCA 2011, pp. 33–56. Springer Berlin Heidelberg (2011)
[15] Maurer, U.: Indistinguishability of random systems. In: Knudsen, L.R. (ed.) Advances in Cryptology – EUROCRYPT 2002. pp. 110–132. Springer Berlin Heidelberg, Berlin, Heidelberg (2002)
[16] Maurer, U., Pietrzak, K., Renner, R.: Indistinguishability amplification. In: Menezes, A. (ed.) Advances in Cryptology – CRYPTO 2007. pp. 130–149. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
[17] Maurer, U., Renner, R.: Abstract cryptography. In: In Innovations in Computer Science – ICS 2011, pp. 1–21. Tsinghua University (2011)
[18] Maurer, U., Renner, R.: From Indifferentiability to Constructive Cryptography (and Back). In: Theory of Cryptography – TCC 2016, pp. 3–24. Springer Berlin Heidelberg (2016)
[19] Nielsen, J.B.: Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. In: Yung, M. (ed.) Advances in Cryptology – CRYPTO. Lecture Notes in Computer Science, vol. 2442, pp. 111–126. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-45708-9_8
[20] Open Whisper Systems. Signal protocol library for java/android. GitHub repository (2017), https://github.com/WhisperSystems/libsignal-protocol-java, accessed: 2018-10-01
[21] Pass, R.: Simulation in quasi-polynomial time, and its application to protocol composition. In: Biham, E. (ed.) Advances in Cryptology — EUROCRYPT 2003. pp. 160–176. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
[22] Pfitzmann, B., Waidner, M.: A model for asynchronous reactive systems and its application to secure message transmission. In: Proceedings 2001 IEEE Symposium on Security and Privacy – S&P 2001. pp. 184–200 (May 2001). https://doi.org/10.1109/SECPRI.2001.924298
[23] Poettering, Bertram and Rösler, Paul: Towards Bidirectional Ratcheted Key Exchange. In: Shacham, H., Boldyreva, A. (eds.) Advances in Cryptology – CRYPTO 2018. pp. 3–32. Springer International Publishing, Cham (2018)
[24] Prabhakaran, M., Sahai, A.: New notions of security: Achieving universal composability without trusted setup. In: Proceedings of the Thirty-sixth Annual ACM Symposium on Theory of Computing. pp. 242–251. STOC '04, ACM, New York, NY, USA (2004). https://doi.org/10.1145/1007352.1007394, http://doi.acm.org/10.1145/1007352.1007394

A Details of Section 5.1

A.1 Key-Updating Signatures

Syntax. A key-updating signature scheme KuSig consists of three polynomial-time algorithms $(\mathsf{KuSig.Gen}, \mathsf{KuSig.Sign}, \mathsf{KuSig.Verify})$. The probabilistic algorithm $\mathsf{KuSig.Gen}$ generates an initial signing key sk and a corresponding verification key vk. Given a message m and sk, the signing algorithm outputs an updated signing key and a signature: $(sk', \sigma) \leftarrow \mathsf{KuSig.Sign}(sk, m)$. Similarly, the verification algorithm outputs an updated verification key and the result v of verification: $(vk', v) \leftarrow \mathsf{KuSig.Verify}(vk, m, \sigma)$.

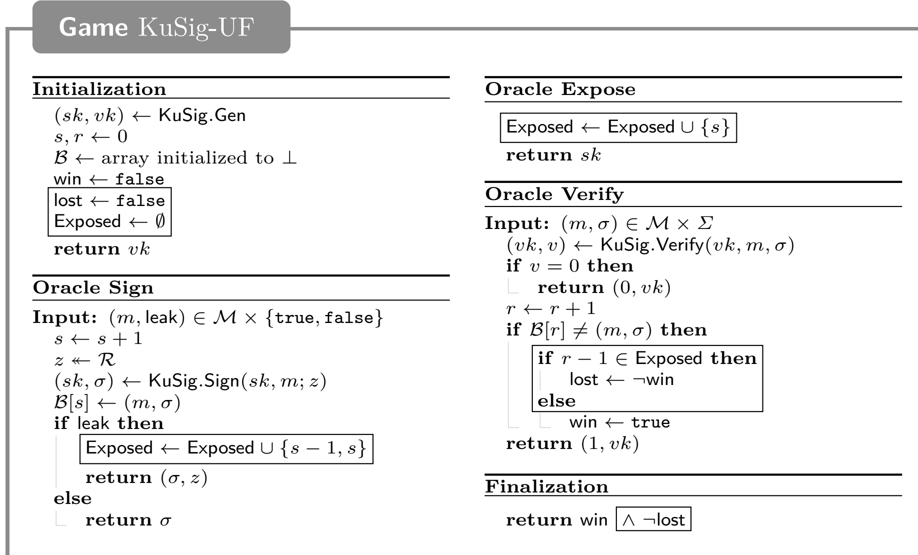

Fig. 14: The strong unforgeability game for key-updating signatures.

Correctness. Let (sk_0, vk_0) be any output of $\mathsf{KuSig.Gen}$, and let $m_1, \dots, m_k$ be any sequence of messages. Further, let $(sk_i, \sigma_i) \leftarrow \mathsf{KuSig.Sign}(sk_{i-1}, m_i)$ and $(vk_i, v_i) \leftarrow \mathsf{KuSig.Verify}(vk_{i-1}, m_i, \sigma_i)$ for $i = 1 \dots (k-1)$. For correctness, we require that $v_i = 1$ for all $i = 1 \dots (k-1)$.

Security. The security of KuSig is formalized using the game KuSig-UF, described in Figure 14. For simplicity, we define the security in the single-user setting (security in the multi-user setting can be obtained using the standard hybrid argument).

A.2 The Authentication Protocol

Recall that in protocol, whenever the sender wants to send a message, a fresh signing and verification key pair is sampled. The fresh verification key is then signed together with the message—using the prior signing key— and the message, the verification key and the signature are transmitted. Finally, the old signing key is securely erased and the fresh one stored instead. Moreover, with each message, the sender also transmits a hash of the previous verification key. The receiver, on the other hand verifies a received message with the previous verification key and stores the new one. See Figure 15 for a formal definition of the two converters.

Converter sig_i

Emulating Interface A of $\mathsf{Ch}^{i-1,\mathtt{A}\to\mathtt{B}}$

Input: $(\mathtt{send}, m, ad) \in \mathcal{M} \times \mathcal{AD}$
assume only called once & isAvailable

// Generate fresh keys
call $r \leftarrow \mathtt{sample}$ at int. A of $\mathsf{Rand}^{kg_i,\mathtt{A}}$
$(sk, vk) \leftarrow \mathsf{Sig.Gen}(r)$

// Send the verification key along the msg
call $(\mathtt{write}, m, (ad, vk))$
at interface A of $\mathsf{Ch}^{i-1,\mathtt{A}\to\mathtt{B}}$

// Store the keys
call $(\mathtt{write}, (sk, vk))$
at interface A of $\mathsf{Mem}^{sk_i,\mathtt{A}}$
return $\mathtt{ok}$

Input: $\mathtt{isAvailable}$
call $succ \leftarrow (\mathtt{isAvailable})$
at interface A of $\mathsf{Ch}^{i-1,\mathtt{A}\to\mathtt{B}}$
return $succ$

Emulating Interface A of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$

Input: $(\mathtt{send}, m, ad) \in \mathcal{M} \times \mathcal{AD}$
assume only called once & isAvailable

// Fetch the signing key
call $(sk, vk) \leftarrow \mathtt{read}$ at int. A of $\mathsf{Mem}^{sk,\mathtt{A}}$

// Sign and send the message
$\sigma \leftarrow \mathsf{Sig.Sign}(sk, (m, ad))$
$h \leftarrow \mathrm{hash}(ad, vk)$
call $(\mathtt{write}, (m, h, \sigma), ad)$
at interface A of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$

// Erase the signing key
call $\mathtt{erase}$ at int. A of $\mathsf{Mem}^{sk_i,\mathtt{A}}$
return $\mathtt{ok}$

Input: $\mathtt{isAvailable}$
call $succ \leftarrow (\mathtt{isAvailable})$
at interface A of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
call $(sk, vk) \leftarrow \mathtt{read}$ at int. A of $\mathsf{Mem}^{sk_i,\mathtt{A}}$
return $succ \wedge ((sk, vk) \neq \bot)$

Converter vrf_i

Emulating Interface B of $\mathsf{Ch}^{i-1,\mathtt{A}\to\mathtt{B}}$

Input: $\mathtt{receive}$
assume only called once

// Receive the message with the vk
call $(m, (ad, vk)) \leftarrow (\mathtt{read})$
at interface B of $\mathsf{Ch}^{i-1,\mathtt{A}\to\mathtt{B}}$
if $(m, (ad, vk)) = \bot$ **then return** $\bot$

// Store the verification key
call $\mathtt{write}, vk$ at int. B of $\mathsf{Mem}^{vk_i,\mathtt{B}}$
return (m, ad)

Input: $\mathtt{isAvailable}$
call $succ \leftarrow (\mathtt{isAvailable})$
at interface B of $\mathsf{Ch}^{i-1,\mathtt{A}\to\mathtt{B}}$
return $succ$

Emulating Interface B of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$

Input: $\mathtt{receive}$
assume only called once & isAvailable

// Receive the message
call $((m, h, \sigma), ad) \leftarrow (\mathtt{read})$
at interface B of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
if $((m, h, \sigma), ad) = \bot$ **then return** $\bot$

// Fetch the verification key
call $vk \leftarrow \mathtt{read}$ at int. B of $\mathsf{Mem}^{vk_i,\mathtt{B}}$

// Verify the signature and hash
$v \leftarrow \mathsf{Sig.Verify}(vk, (m, ad), \sigma)$
$h' \leftarrow \mathrm{hash}(vk, ad)$
if $\neg v \vee h \neq h'$ **then return** $\bot$
return (m, ad)

Input: $\mathtt{isAvailable}$
call $succ \leftarrow (\mathtt{isAvailable})$
at interface B of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
call $vk \leftarrow \mathtt{read}$ at int. A of $\mathsf{Mem}^{vk_i,\mathtt{B}}$
return $succ \wedge (vk \neq \bot)$

Fig. 15: A formal description of the protocol converters sig_i and vrf_i implementing the unidirectional authentication scheme for a single message.

A.3 Proof of Theorem 1

Proof. See Figure 16 for a description of the simulator. Note that the simulator is parameterized in the real world's security guarantees, e.g., the can-leak predicate

of the memory resources. We now proceed to argue that this simulator actually makes the two worlds indistinguishable.

The simulator internally samples a signing-verification key pair and remembers the randomness. Using this, it is easy see to that Alice's memory and randomness resources can be perfectly simulated, since the simulator knows when Alice sent her messages from the corresponding events and the real-world can-leak predicates. Moreover, Bob's memory storing the verification key he receives is also simple to emulate, since the key is transmitted as part of the associated data the simulator sees.

Next, consider the $(i-1)$-st channel. The can-send predicate of the ideal world enforces that the sent event is triggered if and only if it is triggered in the real world. The leakage at Eve's interface is then also simple to simulate: the simulator just appends the verification key to the associated data. Handling injections is also straight-forward: in the real-world, if the distinguisher asks for $(m', (ad', vk'))$ to be injected, then at the point of time Bob fetches this will differ from $(m, (ad, vk))$ if and only one of the components differ and the corresponding injection-function will be evaluated. In the ideal world, the simulator removes vk' and asks to inject (m', ad'), setting the *same* flag to `false` if $vk' \neq vk$, leading to the same behavior. If the distinguisher request a specific error to be triggered, the simulator can simply forward this as well, thereby excluding the signature-verification error.

The interesting part to simulate is everything with respect to the i-th channel. First, observe that Alice will only send the message after sending the $(i-1)$-st, which the can-send predicate in the ideal world ensures as well. Simulating the leakage is also straightforward: the simulator just adds the hash and the signature himself. Now consider a delivery attempt. First, the simulator only processes it once there has also been a delivery on the former channel, such that it knows the verification key Bob will use. Since Bob will not fetch the latter message out-of-order, as enforced by the protocol and the can-receive predicates, respectively, this is not observable, however. Recall that

$$\mathfrak{D}^{\mathsf{S}_i^{\mathsf{auth}}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}(\mathcal{E}, \mathit{same}) := \begin{cases} \mathit{err} & \text{if } \mathfrak{D}^{\mathsf{R}_i^{\mathsf{auth}}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}(\tilde{\mathcal{E}}, \mathit{same}) = \mathit{err} \wedge \mathit{err} \neq \mathtt{msg} \\ \textsf{sig-err} & \text{else if } \neg(\mathit{same} \vee \mathcal{E}_i^{\textsf{sk-known}}) \\ \mathtt{msg} & \text{else} \end{cases}$$

Thus, to show that the two worlds behave identically, we need (1) show that the *same* flag is consistent, as otherwise the channels might trigger different error before it even gets to the signature verification, (2) show that the two worlds trigger a signature-verification error for the same inputs. We consider two cases:

An explicit delivery ($(m', h', \sigma') \neq \mathtt{fwd}$):

1. In order for the ideal-world to behave equivalent, the simulator needs to ensure that the *same* flag in both worlds agree. In the real-world, however, this flag takes the signature and the hash into account, which in the ideal-world are not part of the message but only simulated. The simulator, thus, needs to detect any modification of the signature and the

hash itself, and if necessary enforce $\mathit{same} = \mathsf{false}$. Note that the simulator might never see Alice's message, since the channel might be confidential, and thus cannot trivially check this. It thus proceeds as follows: it verifies the signature using the original verification key (not necessarily the one used by Bob). By correctness of the signature scheme, a failure clearly indicates that something must have been tampered with. By uniqueness of the signatures, it moreover follows that if $(m', ad') = (m, ad)$ and the verification succeeds, then it must have been the same signature. Hence, the simulator can enforce $\mathit{same} = \texttt{false}$ whenever the verification fails, resulting in the channel using $\mathit{same} = \texttt{false}$ iff $(m, \sigma, ad) \neq (m', \sigma', ad')$. For the hash value, the simulator can recompute $\mathrm{hash}(ad', vk)$, and set $\mathit{same} = \mathsf{false}$ whenever it does not match. If $ad = ad'$, this check ensures that $h' = h$, and otherwise the channel will set $\mathit{same} = \mathsf{false}$ anyways. Thus, we know that in the real-world an error happens before the signature verification iff the same happens in the idea-world.

2. Now consider the signature verification and the hash check. For the former, the simulator simply verifies the signature using vk' (which Bob uses in the real-world). If the check succeeds, it injects the message. Otherwise, it triggers a signature-failure event. It remains to see that whenever the simulator tries to inject the message this is actually allowed by $\mathfrak{D}^{\mathsf{S}_i^{\mathsf{auth}}}_{\mathsf{Ch}(i,\mathtt{A}\rightarrow\mathtt{B})}(\mathcal{E}, \mathit{same})$. It is, however, easy to see that this happening would directly imply an existential forgery, since it means that Eve injected a different message with a valid signature, with respect to the correct verification key, and without having any information about the signing key (neither the key nor its randomness leaked). The simulator can furthermore easily check that $h' = \mathrm{hash}(ad', vk')$, this time using vk', which also Bob would use in the real-world.

A forwarding request: Whenever the distinguisher asks to deliver the original message, but with a potentially different associated data, the *same* flag is trivially to decide: its the same iff the associated data match. Since this is what the channel checks anyways, there is nothing for the simulator to be taken care of. Now consider the check performed by Bob's protocol. In the real world, by collision resistance, the hashes will match iff $ad = ad'$ and $vk = vk'$. In this case, by correctness, moreover the signature verification will also succeed. Hence, the protocol accepts iff $ad = ad'$ and $vk = vk'$, which can easily be emulated by the simulator.

If the distinguisher asks to trigger an error, this can be handled analogous to the delivery request. More precisely, the simulator decides *same* analogously and then requests an the error to be triggered, thereby excluding the signature error.

Simulator $\mathsf{sim}_{\mathsf{auth}}$

Let ℓ_{sig} and ℓ_{hash} the length of a signature and a hash, respectively.

Initialization

$r \twoheadleftarrow \mathcal{R}$
$(sk, vk) \leftarrow \mathsf{Sig.Gen}(r)$
$vk' \leftarrow \bot$

Emulating Interface $\mathtt{E}$ **of** $\mathsf{Mem}^{sk_i,\mathtt{A}}$

Input: `read`
  if $\mathcal{E}^{\mathsf{sent}}_{\mathsf{Ch}(i-1,\mathtt{A}\to\mathtt{B})} \land \neg\mathcal{E}^{\mathsf{sent}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})} \land \mathfrak{L}^{\mathsf{R}_i^{\mathsf{auth}}}_{\mathsf{Mem}(sk_i,\mathtt{A})}(\tilde{\mathcal{E}})$ **then**
    $\mathcal{E} \xleftarrow{+} \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Mem}(sk_i,\mathtt{A})}$
    return (sk, vk)
  else
    return $\bot$

Emulating Interface $\mathtt{E}$ **of** $\mathsf{IMem}^{vk_i,\mathtt{B}}$

Input: `read`
  if $\mathcal{E}^{\mathsf{received}}_{\mathsf{Ch}(i-1,\mathtt{A}\to\mathtt{B})}$ **then**
    return vk'
  else
    return $\bot$

Emulating Interface $\mathtt{E}$ **of** $\mathsf{Rand}^{kg_i,\mathtt{A}}$

Input: `triggerLeaking`
  if $\neg\mathfrak{L}^{\mathsf{R}^{\mathsf{auth}}}_{\mathsf{Rand}(kg_i,\mathtt{A})}(\tilde{\mathcal{E}})$ **then return** $\bot$
  $\mathcal{E} \xleftarrow{+} \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Rand}(kg_i,\mathtt{A})}$
  return `ok`

Input: `getLeakage`
  if $\neg\mathcal{E}^{\mathsf{sent}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})} \lor \neg\mathcal{E}^{\mathsf{leaked}}_{\mathsf{Rand}(kg_i,\mathtt{A})}$ **then**
    return $\bot$
  return r

Emulating Interface $\mathtt{E}$ **of** $\mathsf{Ch}^{i-1,\mathtt{A}\to\mathtt{B}}$

Input: `read`
  call $(m, ad) \leftarrow (\mathtt{read})$
    at interface E of $\mathsf{Ch}^{i-1,\mathtt{A}\to\mathtt{B}}$
  if $(m, ad) = \bot$ **then return** $\bot$
  else return $(m, (ad, vk))$

Input: `readLength`
  call $(\ell, ad) \leftarrow (\mathtt{readLength})$
    at interface E of $\mathsf{Ch}^{i-1,\mathtt{A}\to\mathtt{B}}$
  if $(\ell, ad) = \bot$ **then return** $\bot$
  else return $(\ell, (ad, vk))$

Input: $(\mathtt{deliver}, m', (ad', vk''), \mathit{same})$
  $vk' \leftarrow vk''$
  if $\mathit{same} = \mathtt{check} \land vk'' \neq vk$ **then**
    $\mathit{same} \leftarrow \mathtt{false}$
  call $(\mathtt{deliver}, m', ad', \mathit{same})$
    at interface E of $\mathsf{Ch}^{i-1,\mathtt{A}\to\mathtt{B}}$
  return `ok`

Input: $(\mathtt{error}, \mathit{err}, m', (ad', vk''), \mathit{Overw}, \mathit{same})$
  if $\mathit{same} = \mathtt{check} \land vk'' \neq vk$ **then**
    $\mathit{same} \leftarrow \mathtt{false}$
  call $(\mathtt{error}, \mathit{err}, \mathit{Overw} \cup \{\mathsf{sig\text{-}err}\}, m', ad', \mathit{same})$
    at interface E of $\mathsf{Ch}^{i-1,\mathtt{A}\to\mathtt{B}}$
  return `ok`

Emulating Interface $\mathtt{E}$ **of** $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$

Input: $(\mathtt{deliver}, (m', h', \sigma'), ad', \mathit{same})$
  if $(m', h', \sigma'), = \mathtt{fwd} \land \neg\mathcal{E}^{\mathsf{sent}}_{\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}}$ **then**
    return $\bot$

  At this point, if delivery of $(i-1)$-st channel not called, then output `ok` immediately but delay processing until then.

  if $(m', h', \sigma') = \mathtt{fwd}$ **then**
    call $(\ell, ad) \leftarrow (\mathtt{readLength})$
      at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
    if $ad' = ad \land vk' = vk$ **then**
      call $(\mathtt{deliver}, \mathtt{fwd}, ad', \mathit{same})$
        at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
    else
      if $\mathit{same} = \mathtt{check}$ **then**
        $\mathit{same} \leftarrow ad' = ad$
      call $(\mathtt{error}, \mathsf{sig\text{-}err}, \emptyset, \bot, \bot, \mathit{same})$
        at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
  else
    $v_{\mathsf{S}} \leftarrow \mathsf{Sig.Verify}(vk, (m', ad'), \sigma)$
    $v_{\mathsf{R}} \leftarrow \mathsf{Sig.Verify}(vk', (m', ad'), \sigma')$
    $h_{\mathsf{S}} \leftarrow \mathrm{hash}(ad, vk)$
    $h_{\mathsf{R}} \leftarrow \mathrm{hash}(ad', vk)$
    if $\mathit{same} = \mathtt{check}$ **then**
      if $\neg v_{\mathsf{S}} \lor h_{\mathsf{S}} \neq h'$ **then**
        $\mathit{same} \leftarrow \mathtt{false}$
    if $v_{\mathsf{R}} \land h_{\mathsf{R}} = h'$ **then**
      call $(\mathtt{deliver}, m', ad', \mathit{same})$
        at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
    else
      call $(\mathtt{error}, \mathsf{sig\text{-}err}, \emptyset, m', ad', \mathit{same})$
        at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
  return `ok`

Input: $(\mathtt{error}, \mathit{err}, \mathit{Overw}, (m', h', \sigma'), ad', \mathit{same})$
  if $\mathit{same} = \mathtt{check}$ **then**
    handle as in deliver by computing v_{S} and h_{S}
  $\mathit{Overw} \leftarrow \mathit{Overw} \cup \{\mathsf{sig\text{-}err}\}$
  call $(\mathtt{error}, \mathit{err}, \mathit{Overw}, m', ad', \mathit{same}))$
    at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
  return `ok`

Input: `read`
  call $(m, ad) \leftarrow (\mathtt{read}, m, ad)$
    at interface E of $\mathsf{Ch}^{i-1,\mathtt{A}\to\mathtt{B}}$
  if $(m, ad) = \bot$ **then return** $\bot$
  $\sigma \leftarrow \mathsf{Sig.Sign}(sk, (m, ad))$
  return $((m, h(vk, ad), \sigma), ad)$

Input: `readLength`
  call $(\ell, ad) \leftarrow (\mathtt{readLength})$
    at interface E of $\mathsf{Ch}^{i-1,\mathtt{A}\to\mathtt{B}}$
  if $(\ell, ad) = \bot$ **then return** $\bot$
  return $(\ell + \ell_{\mathsf{sig}} + \ell_{\mathsf{hash}}, ad)$

Fig. 16: The simulator for Theorem 1.

B Details of Section 5.2

B.1 The Sesqui-directional HIBE Protocol

Recall that the protocol proceeds in epochs, where each epoch is initiated by Bob sending a fresh HIBE public key *mpk* (and indicating how many messages he received at this moment). Within the epoch, Alice sends then a sequence of messages to Bob, encrypted under this public key and using as identity (the hashes of) all ciphertexts she sent since the message indicated by Bob. See Figure 17 for a schematic depiction of the scheme.

See Figure 18 for a formal definition of the two converters hibe-enc and hibe-dec, respectively, that implement this protocol when connected to the real-world resource $\mathsf{R}^{\mathsf{hibe}}$. Note that in the formal definition we allow ourselves to be a bit sloppy when it comes to determine in which epoch a party is, or how many messages a party already sent or received. While this in principle can be decuded from the memory resources the parties have available, a reasonable implementation would of course just store this (public) information directly.

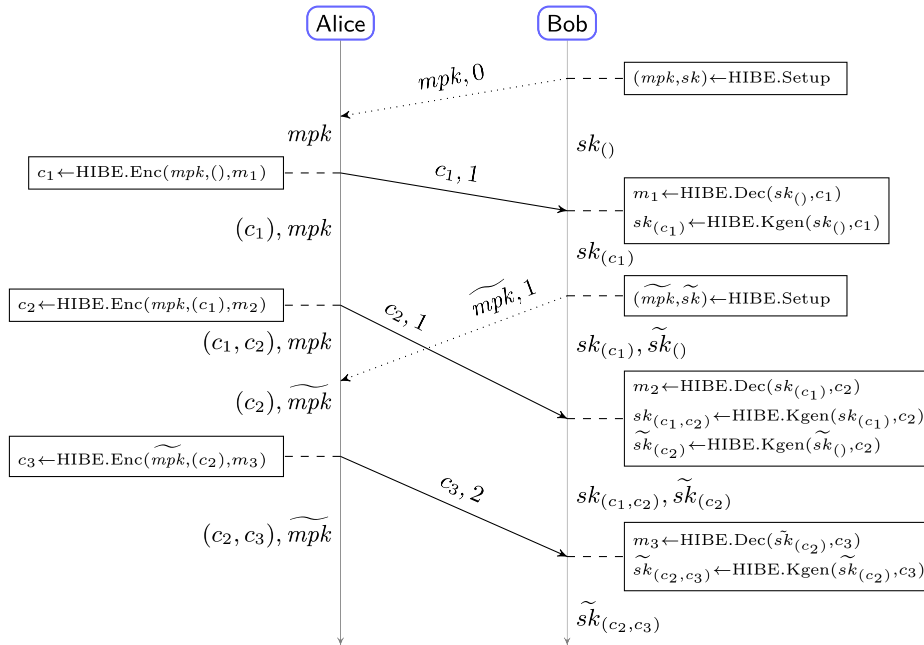

Fig. 17: Sesqui-directional confidentiality from HIBE, depicting the first and the beginning of the second epoch.

Converter hibe-enc

Emulating Interface A of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$, $i \in [n]$

Input: $(\mathtt{send}, m, ad) \in \mathcal{M} \times \mathcal{AD}$
  assume only called once & isAvailable

  // Fetch the mpk
  call $(j, mpk, rcv) \leftarrow (\mathtt{read})$
    at interface A of $\mathsf{IMem}^{pk,\mathtt{A}}$

  // Compute the identity
  $tr \leftarrow ()$
  for $k = (rcv+1), \ldots, i-1$ **do**
    call $h_k \leftarrow \mathtt{read}$ at int. A of $\mathsf{Mem}^{tr_k,\mathtt{A}}$
    $tr \leftarrow tr \parallel h_k$

  // Encrypt and send
  call $r \leftarrow \mathtt{sample}$ at int. A of $\mathsf{Rand}^{enc_i,\mathtt{A}}$
  $c \leftarrow \text{HIBE.Enc}(mpk, tr, m; r)$
  call $succ \leftarrow (\mathtt{write}, c, (j, ad))$
    at interface A of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$

  // Store the hash
  call $((\mathtt{write}, \text{hash}(c, j, ad)))$
    at interface A of $\mathsf{Mem}^{tr_i,\mathtt{A}}$
  return ok

Input: isAvailable
  call $succ \leftarrow (\mathtt{isAvailable})$
    at interface A of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
  call $(j, mpk, rcv) \leftarrow (\mathtt{read})$
    at interface A of $\mathsf{IMem}^{pk,\mathtt{A}}$
  $succ \leftarrow succ \wedge ((j, mpk, rcv) \neq \bot)$
  if $i > 1$ **then**
    call $tr \leftarrow \mathtt{read}$ at int. A of $\mathsf{Mem}^{tr_{i-1},\mathtt{A}}$
    $succ \leftarrow succ \wedge (tr \neq \bot)$
  return $succ$

Emulating Interface A of $\mathsf{Ch}^{j,\mathtt{B}\to\mathtt{A}}$, $j \in [n]$

Input: receive
  assume only called once

  // Read the message
  call $(m, (mpk, rcv, ad)) \leftarrow (\mathtt{read})$
    at interface A of $\mathsf{Ch}^{j,\mathtt{B}\to\mathtt{A}}$
  if $(m, (mpk, r, ad)) = \bot$ **then**
    return $\bot$

  // Store the mpk
  call $(\mathtt{write}, (j, mpk, rcv))$
    at interface A of $\mathsf{IMem}^{pk,\mathtt{A}}$
  return (m, ad)

Input: isAvailable
  call $succ \leftarrow (\mathtt{isAvailable})$
    at interface A of $\mathsf{Ch}^{j,\mathtt{B}\to\mathtt{A}}$
  call $(j', mpk, rcv) \leftarrow (\mathtt{receive})$
    at interface A of $\mathsf{IMem}^{pk,\mathtt{A}}$
  return $succ \wedge (j', mpk, rcv) \neq \bot$
      $\wedge\, j = j' + 1$

Converter hibe-dec

Emulating Interface B of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$, $i \in [n]$

Input: receive
  assume only called once & isAvailable
  call $(c, (ep, ad)) \leftarrow (\mathtt{read})$
    at interface A of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$

  // Valid epoch?
  Let e_{max} be max s.t. $\mathsf{Mem}^{sk(e_{max},i),\mathtt{B}} \neq \bot$
  Let e_{min} be min s.t. $\mathsf{Mem}^{sk(e_{min},i),\mathtt{B}} \neq \bot$
  if $ep = \bot \vee \neg(e_{min} \leq ep \leq e_{max})$ **then**
    $valid \leftarrow \mathtt{false}$

  // Decrypt
  call $sk \leftarrow \mathtt{read}$ at int. B of $\mathsf{Mem}^{sk_{ep,i},\mathtt{B}}$
  if $sk \neq \bot \wedge valid$ **then**
    $m \leftarrow \text{HIBE.Dec}(sk, c)$

  // Update the keys
  if $valid \wedge m \neq \bot$ **then**
    $h \leftarrow \text{hash}(c, ep, ad)$
    for $k = ep, \ldots, e_{max}$ **do**
      call $sk \leftarrow (\mathtt{read})$
        at interface B of $\mathsf{Mem}^{sk(k,i),\mathtt{B}}$
      $sk' \leftarrow \text{HIBE.Kgen}(sk, h)$
      call $(\mathtt{write}, sk')$ at int. B of $\mathsf{Mem}^{sk(k,i+1),\mathtt{B}}$
  for $k = e_{min}, \ldots, e_{max}$ **do**
    call erase at int. B of $\mathsf{Mem}^{sk(k,i),\mathtt{B}}$

  if $valid \wedge m \neq \bot$ **then return** (m, ad)
  else return $\bot$

Input: isAvailable
  call $succ \leftarrow (\mathtt{isAvailable})$
    at interface B of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
  return true iff $succ$ and there exists j
    s.t. $\mathsf{Mem}^{sk(j,i),\mathtt{B}} \neq \bot$

Emulating Interface B of $\mathsf{Ch}^{j,\mathtt{B}\to\mathtt{A}}$, $j \in [n]$

Input: $(\mathtt{send}, m, ad) \in \mathcal{M} \times \mathcal{AD}$
  assume only called once & isAvailable

  // Number of received messages
  Let i be max s.t. $\mathsf{Mem}^{sk(j-1,i+1),\mathtt{B}} \neq \bot$
  if $j > 1$ and 0 otherwise.

  // Generate the (mpk,msk) pair
  call $r \leftarrow \mathtt{sample}$ at int. B of $\mathsf{Rand}^{kg,\mathtt{B}}$
  $(mpk, msk) \leftarrow \text{HIBE.Setup}(r)$
  call $(\mathtt{write}, m, (mpk, i, ad))$
    at interface A of $\mathsf{Ch}^{k,\mathtt{B}\to\mathtt{A}}$
  call $(\mathtt{write}, msk)$
    at interface B of $\mathsf{Mem}^{sk(j,i+1),\mathtt{B}}$
  return ok

Input: isAvailable
  call $succ \leftarrow (\mathtt{isAvailable})$
    at interface B of $\mathsf{Ch}^{j,\mathtt{B}\to\mathtt{A}}$
  return true iff $succ$ and $(j-1)$-st sent

Fig. 18: A formal description of the converters hibe-enc and hibe-dec for the sender and receiver, respectively.

B.2 Proof of Theorem 2

We require the HIBE scheme to be IND-CCA secure with the following (non-standard) additional properties: first, we require that decrypting a honestly generated ciphertext with an unauthorized decryption key (further down or in a different path in the hierarchy) not only decrypts to something unrelated, but fails to decrypts except with negligible probability.[9] Second, we require encryption to be truly randomized, that is, even given the secret key, no adversary can output a message-ciphertext pair such that a fresh encryption of the message results in that ciphertext, except with negligible probability.[10]

Proof. A formal description of the simulator $\mathsf{sim}_{\mathsf{hibe}}$ is given below. The simulator initially samples all randomness, for the key generation and the encryption, and generates all the master key pairs. Using this, it is easy to simulate the randomness resources $\mathsf{Rand}^{kg_j,\mathtt{B}}$, $\mathsf{Rand}^{enc_i,\mathtt{A}}$, as well as the memories storing the master secret keys. Furthermore, simulating the channels for transporting the master public keys, and the memory $\mathsf{Mem}^{pk,\mathtt{A}}$ storing them, is trivial as well. The other memories $\mathsf{Mem}^{sk_{(j,i)},\mathtt{B}}$ and $\mathsf{Mem}^{tr_i,\mathtt{A}}$ storing the derived secret keys and the ciphertext hashes, respectively, can be easily simulated once we can consistently simulate the ciphertexts.

Observe that the simulator needs to generate ciphertexts in the following situations:

- The distinguisher explicitly queries the i-th ciphertext, or the hash thereof.
- The distinguisher asks for the i-th secret key of the j-th epoch: In this situation the simulator needs to generate all ciphertexts, up to the first injection, with which the receiver updated the j-th master secret key. After altering the keys for the j-th epoch with an injection, we know from our additional assumption that forwarding the correct ciphertext will cause a decryption error, so Eve needs to provide her own, which are then used to compute the secret keys.
- The distinguisher injects the first message in an epoch. At this point, the secret key is still in sync and the simulator needs to decrypt under that key to determine the message.

To simulate the ciphertexts we follow the usual strategy: if the message is known to the simulator at the time of simulating the ciphertext, then he simply encrypts it. Otherwise, if only the length is known, he encrypts the string of zeros of the same length instead. By the CCA-security of the HIBE scheme, this is indistinguishable unless after encrypting the fake message the simulator has to reveal an earlier secret-key for that epoch (up to an injection) that allows for trivial decryption. Note that keys after an injection do not reveal anything about the encryptions, since the identities do not match.

[9] This can be achieved by including sufficient redundancy.
[10] For instance, IND-CCA allows the public-key to encode a message for which encryption is deterministic, as long as one cannot devise that message from the public key.

Our workaround for the commitment issue, however, prevents fake ciphertexts to be exposed after each of the situation in which the simulator had to produce one. In addition, note that whenever the encryption randomness leaked, the simulator gets the real message immediately, due to our additional assumption in the real world, the simulator is allowed to fetch the message, without triggering an event, as soon as the channel becomes insecure. Hence, he can simply encrypt the correct message and avoid the commitment issue.

Finally, consider how delivering on the channels is handled by the simulator. First, observe that the simulator can process the channels in order, i.e, delay handling them until all previous ones have been handled, since Bob will only fetch them in order. To actually handle the i-th delivery request, the simulator proceeds as follows: firstly, he determines whether the ciphertext, epoch number, and associated data that Eve wants to deliver are the same as Alice sent. For the ciphertext we can use the following strategy:

- If Alice did not send her message yet, then she also did not sample the corresponding randomness. Hence, by our additional assumption we know that it won't be the same ciphertext.
- If Alice sent her message and Eve requests to forward it, it is trivially the same.
- If Alice sent her message, and either we already simulated the ciphertext—or the message is not secret and we can simply create the real one now—then we can simply compare.
- If Alice sent her message but it is still secret (in particular, the randomness did not leak yet) and Eve did not see the simulated ciphertext yet, then it is a different one, since again the randomness is unknown to Eve.

If the ciphertext matches, then we can also simply check the epoch number and associated data.

Once the simulator knows whether Eve's message matches Alice's, it knows whether Bob will update his secret keys with the correct identity. Hence, we especially known after processing each delivery request whether at the end Bob will still have the correct decryption key (without having to generate them) or whether the parties are now out of sync. Based on this information, the simulator proceeds as follows:

- If it is the same ciphertext, and Bob will use the correct decryption key (i.e., correct epoch and key of this epoch has been updated in sync), issue a forward command for the message. By correctness of the scheme, this simulates decryption correctly.
- If it is the same ciphertext, but Bob uses the wrong secret key, then issue an decryption error. By our stronger assumption, this simulates the real-world behavior.
- If it is a different ciphertext, the simulator generates the corresponding decryption key, performs the decryption, and either injects the message or triggers the decryption error, depending on the outcome.

It remains to argue that the simulator also works correctly if Eve tampered the transmission of the master public key, or the associated value indicating how many messages the receiver already obtained when creating that key. In this situation, all messages sent by Alice are treated as insecure and the simulator, thus, can simulate the actual ciphertexts. For delivery request, not that modifying either value will lead to Alice encrypting for the wrong key: either the master public key or the identity does not match. We can consider two situations: if the request happens after it is clear that Alice uses a wrong public key or identity, then the parties are treated as out of sync. If Eve forwards a ciphertext, then by our assumption, Bob's decryption will fail, which is what the simulator replicates. For newly injected ciphertexts, the simulator anyway always replicates the correct decryption. If Eve request a delivery at which point it is unclear yet whether Alice uses the correct public key, then the ciphertext will be treated as an injection, and the simulator just replicates Bob's behavior of the real world. □

Simulator $\mathsf{sim}_{\mathsf{hibe}}$

Let HibeEncLen(ℓ, d) denote the function determining the ciphertext length based on the message length ℓ and the hierarchy level d.

Initialization

$C, C', AD', TR, MPK, MPK', MSK, SK, E', R', InSync \leftarrow$ array init. to $\bot$
$(r^1_{kg}, \dots, r^n_{kg}) \twoheadleftarrow \mathcal{R}^n$
$(r^1_{enc}, \dots, r^n_{enc}) \twoheadleftarrow \mathcal{R}^n$
for $j \in [n]$ **do**
  $(MSK[j], MPK[j]) \leftarrow$ HIBE.Kgen(r^j_{kg})

Emulating Interface E of $\mathsf{IMem}^{pk,\mathtt{A}}$

Input: `read`
  Let j max s.t. $\mathcal{E}^{\mathsf{received}}_{\mathsf{Ch}(j,\mathtt{B}\to\mathtt{A})}$ and $\neg\mathcal{E}^{\mathsf{received}}_{\mathsf{Ch}(j+1,\mathtt{B}\to\mathtt{A})}$
  if no such j exists **then**
    return $\bot$
  else
    return $(j, MPK'[j], R'[j])$

Emulating Interface E of $\mathsf{Mem}^{tr_i,\mathtt{A}}$, $i \in [n]$

Input: `read`
  if $\neg\mathcal{E}^{\mathsf{sent}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})} \vee \neg\mathfrak{L}^{\mathsf{R}^{\mathsf{hibe}}}_{\mathsf{Mem}(tr_i,\mathtt{A})}(\tilde{\mathcal{E}})$ **then**
    return $\bot$
  if $TR[i] = \bot$ **then**
    CreateTranscript(i)
  $\mathcal{E} \xleftarrow{+} \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Mem}(tr_i,\mathtt{A})}$
  return $TR[i]$

Emulating Interface E of $\mathsf{Mem}^{sk(j,i),\mathtt{B}}$, $j \in [n], i \in [n+1]$

Input: `read`
  if $\mathcal{E}^{\mathsf{received}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})} \vee \mathcal{E}^{\mathsf{error}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})} \vee \neg\mathfrak{L}^{\mathsf{R}^{\mathsf{hibe}}}_{\mathsf{Mem}(sk_{(j,i)},\mathtt{B})}(\tilde{\mathcal{E}})$ **then**
    return $\bot$
  CreateSK(j, i)
  if $SK[j, i] \neq \bot$ **then**
    $\mathcal{E} \xleftarrow{+} \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Mem}(sk_i,\mathtt{B})}$
  return $SK[j, i]$

Emulating Interface E of $\mathsf{Rand}^{kg_j,\mathtt{B}}$, $j \in [n]$

Input: `triggerLeaking`
  if $\neg\mathfrak{L}^{\mathsf{R}^{\mathsf{hibe}}}_{\mathsf{Rand}(kg_j,\mathtt{B})}(\tilde{\mathcal{E}})$ **then return** $\bot$
  $\mathcal{E} \xleftarrow{+} \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Rand}(kg_j,\mathtt{B})}$
  return `ok`

Input: `getLeakage`
  if $\neg\mathcal{E}^{\mathsf{sent}}_{\mathsf{Ch}(j,\mathtt{B}\to\mathtt{A})} \vee \neg\mathcal{E}^{\mathsf{leaked}}_{\mathsf{Rand}(kg_j,\mathtt{B})}$ **then**
    return $\bot$
  return r^j_{kg}

Emulating Interface E of $\mathsf{Rand}^{enc_i,\mathtt{A}}$, $i \in [n]$

Input: `triggerLeaking`
  if $\neg\mathfrak{L}^{\mathsf{R}^{\mathsf{hibe}}}_{\mathsf{Rand}(enc_i,\mathtt{A})}(\tilde{\mathcal{E}})$ **then return** $\bot$
  $\mathcal{E} \xleftarrow{+} \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Rand}(enc_i,\mathtt{A})}$
  return `ok`

Input: `getLeakage`
  if $\neg\mathcal{E}^{\mathsf{sent}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})} \vee \neg\mathcal{E}^{\mathsf{leaked}}_{\mathsf{Rand}(enc_i,\mathtt{A})}$ **then**
    return $\bot$
  return r^i_{enc}

Emulating Interface E of $\mathsf{Ch}^{j,\mathtt{B}\to\mathtt{A}}$, $j \in [n]$

Input: `read`
- **call** $(m, ad) \leftarrow$ `read` at int. E of $\mathsf{Ch}^{k,\mathtt{B}\to\mathtt{A}}$
- **if** $(m, ad) = \bot$ **then return** $\bot$
- **else**
  - $r \leftarrow \textsc{Received}(j)$
  - **return** $(m, (MPK[j], r, ad))$

Input: `readLength`
- **call** $(\ell, ad) \leftarrow$ (`readLength`) at interface E of $\mathsf{Ch}^{k,\mathtt{B}\to\mathtt{A}}$
- **if** $(\ell, ad) = \bot$ **then return** $\bot$
- **else**
  - $r \leftarrow \textsc{Received}(j)$
  - **return** $(\ell, (MPK[j], r, ad))$

Input: $(\mathtt{deliver}, m, (mpk', r', ad), same)$
- $MPK'[j] \leftarrow mpk'$
- $R'[j] \leftarrow r'$
- $r \leftarrow \textsc{Received}(j)$
- **if** $(mpk', r') = (MPK[j], r)$ **then**
  - $InSync[j, r] \leftarrow \mathtt{true}$
- **else**
  - $same \leftarrow \mathtt{false}$
  - $InSync[j, r] \leftarrow \mathtt{false}$
- **call** $(\mathtt{deliver}, m, ad, same)$ at interface E of $\mathsf{Ch}^{k,\mathtt{B}\to\mathtt{A}}$
- **return** `ok`

Input: $(\mathtt{error}, err, Overw, m, (mpk', r', ad), same)$
- $r \leftarrow \textsc{Received}(j)$
- **if** $same = \mathtt{check} \wedge (mpk', r') \neq (MPK[j], r)$ **then**
  - $same \leftarrow \mathtt{false}$
  - $InSync[j, r] \leftarrow \mathtt{false}$
- **call** $(\mathtt{error}, err, Overw, m, ad, same)$ at interface E of $\mathsf{Ch}^{k,\mathtt{B}\to\mathtt{A}}$
- **return** `ok`

Emulating Interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$, $i \in [n]$

Input: `read`
- **if** $\mathfrak{L}^{\mathsf{R}^{\mathsf{hibe}}}_{\mathsf{Chan}(i,\mathtt{A}\to\mathtt{B})} = \mathtt{false}$ **then**
  - **return** $\bot$
- **else if** $\mathfrak{L}^{\mathsf{R}^{\mathsf{hibe}}}_{\mathsf{Chan}(i,\mathtt{A}\to\mathtt{B})} = \mathtt{true}$ **then**
  - $\mathcal{E} \xleftarrow{+} \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Chan}(i,\mathtt{A}\to\mathtt{B})}$
- **call** $(\ell, ad) \leftarrow$ (`readLength`) at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
- $e \leftarrow \textsc{Epoch}(i)$
- **if** $C[i] = \bot$ **then**
  - $\textsc{CreateCiphertext}(i)$
- **return** $(C[i], (e, ad))$

Input: `readLength`
- **call** $(\ell, ad) \leftarrow$ (`readLength`) at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
- **if** $(\ell, ad) = \bot$ **then return** $\bot$
- $e \leftarrow \textsc{Epoch}(i)$
- **return** $(\mathrm{HibeEncLen}(\ell, i - 1 - R'[e]), (e, ad))$

Int E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$ cont.

Input: $(\mathtt{deliver}, c', (e', ad'), same)$
- **if** $c' = \mathtt{fwd} \wedge \neg\mathcal{E}^{\mathsf{sent}}_{\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}}$ **then**
  - **return** $\bot$

At this point, if delivery of $(i-1)$-st channel not called, then output `ok` immediately but delay processing until then.

- $(E'[i], AD'[i], C'[i]) \leftarrow (e', ad', c')$
- **if** $\neg\textsc{ValidEpoch}(e', i)$ **then**
  - **call** $(\mathtt{error}, \mathsf{dec\text{-}err}, \emptyset, same)$ at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
  - **return** `ok`
- // Same ciphertext?
- **call** $(\ell, (e, ad)) \leftarrow$ (`readLength`) at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
- **if** $(\ell, (e, ad)) = \bot$ **then**
  - $sameC \leftarrow \mathtt{false}$
- **else if** $c' = \mathtt{fwd}$ **then**
  - $sameC \leftarrow \mathtt{true}$
- **else if** $C[i] \neq \bot \vee \mathfrak{L}^{\mathsf{S}^{\mathsf{hibe}}}_{\mathsf{Chan}(i,\mathtt{A}\to\mathtt{B})}(\mathcal{E}) = \mathtt{silent}$ **then**
  - **if** $C[i] = \bot$ **then**
    - $\textsc{CreateCiphertext}(i)$
  - $sameC \leftarrow (C[i] = c')$
- **else**
  - $sameC \leftarrow \mathtt{false}$
- // Overall: injection?
- $same' \leftarrow sameC \wedge (e', ad') = (e, ad)$
- $InSync[e', i] \leftarrow InSync[e', i-1] \wedge same'$
- **if** $\neg(same')$ **then** $same \leftarrow \mathtt{false}$
- // Handle request
- **if** $sameC$ **then**
  - **if** $e = e' \wedge InSync[e', i-1]$ **then**
    - **call** $(\mathtt{deliver}, \mathtt{fwd}, ad', same)$ at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
  - **else**
    - **call** $(\mathtt{error}, \mathsf{dec\text{-}err}, \emptyset, same)$ at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
- **else**
  - **if** $SK[e', i] = \bot$ **then**
    - $\textsc{CreateSK}(e', i)$
  - $m' \leftarrow \mathrm{HIBE.Dec}(SK[e', i], c')$
  - **if** $m' \neq \bot$ **then**
    - **call** $(\mathtt{deliver}, m', ad', same)$ at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
  - **else**
    - **call** $(\mathtt{error}, \mathsf{dec\text{-}err}, \emptyset, same)$ at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
- **return** `ok`

Input: $(\mathtt{error}, err, Overw, c', (e', ad'), same)$
- **if** $same = \mathtt{check}$ **then**
  - determine $same'$ as in for handling the `deliver` command above.
  - $same \leftarrow same'$ // `true` or `false`
- $Overw \leftarrow Overw \cup \{\mathsf{dec\text{-}err}\}$
- $m \twoheadleftarrow \mathcal{M}$ // ignored
- **call** $(\mathtt{error}, err, Overw, m, ad', same)$ at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
- **return** `ok`

Function Epoch(i)

$e = \max\{j \in [n] \mid \mathcal{E}^{\mathsf{received}}_{\mathsf{Ch}(j,\mathtt{B}\to\mathtt{A})} \prec \mathcal{E}^{\mathsf{sent}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})}\}$
return e

Procedure ValidEpoch(e, i), $e, i \in [n]$

$rcv \leftarrow$ Received(e)
return $\mathcal{E}^{\mathsf{received}}_{\mathsf{Ch}(i-1,\mathtt{A}\to\mathtt{B})} \wedge \mathcal{E}^{\mathsf{sent}}_{\mathsf{Ch}(e,\mathtt{B}\to\mathtt{A})} \wedge rcv < i \wedge E'[i-1] \leq e$

Function Received(j)

// Number of msg. Bob received when sending the j-th msg.
$r = \max\Big(\{i \in [n] \mid \mathcal{E}^{\mathsf{received}}_{\mathsf{Ch}(i,\mathtt{A}\to\mathtt{B})} \prec \mathcal{E}^{\mathsf{sent}}_{\mathsf{Ch}(j,\mathtt{B}\to\mathtt{A})}\} \cup \{0\}\Big)$
return r

Procedure CreateCiphertext(i), $i \in [n]$

$e \leftarrow$ Epoch(i)
$id \leftarrow ()$
for $k = (R'[e]+1), \ldots, (i-1)$ **do**
  if $TR[k] = \bot$ **then**
    CreateTranscript(k)
  $id \leftarrow id \parallel TR[k]$
call $(m, ad) \leftarrow$ `read` at int. E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
if $(m, ad) = \bot$ **then**
  call $(\ell, ad) \leftarrow$ (`readLength`) at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
  $m \leftarrow 0^{\ell}$
$C[i] \leftarrow$ HIBE.Enc$(MPK'[e], id, m; r^i_{enc})$

Procedure CreateTranscript(i), $i \in [n]$

if $C[i] = \bot$ **then**
  CreateCiphertext(i)
call $(\ell, ad) \leftarrow$ (`readLenght`) at interface E of $\mathsf{Ch}^{i,\mathtt{A}\to\mathtt{B}}$
$TR[i] \leftarrow$ hash$(C[i],$ Epoch$(i), ad)$

Procedure CreateSK(j, i)

$rcv \leftarrow$ Received(j)
if $SK[j,i] \neq \bot \vee \neg$ValidEpoch$(j,i)$ **then**
  return
else if $i = rcv + 1$ **then**
  $SK[j,i] \leftarrow MSK[j]$
else
  CreateSK$(j, i-1)$
  if $C'[i-1] =$ `fwd` **then**
    if $C[i-1] = \bot$ **then**
      CreateCiphertext$(i-1)$
    $C'[i-1] \leftarrow C[i-1]$
  $h \leftarrow$ hash$(C'[i-1], E'[i-1], AD'[i-1])$
  $SK[j,i] \leftarrow$ HIBE.Kgen$(SK[j,i-1], h)$