A Fresh Approach to Updatable Symmetric Encryption

Andrés Fabrega[1], Ueli Maurer[2], and Marta Mularczyk[2⋆]

[1] andresfg@mit.edu, MIT
[2] {maurer,mumarta}@inf.ethz.ch, ETH Zurich

Abstract. Updatable encryption (UE) is symmetric encryption which additionally supports key rotation. UE was introduced for scenarios where a user stores encrypted data on a cloud and, in order to mitigate secret key leakage, periodically sends a short update token, which the cloud uses to re-encrypt stored data to a fresh key. A long line of research resulted in a wide variety of security properties UE schemes can provide, including confidentiality, integrity protection, and hiding metadata. Unfortunately, given the complexity and nuances in the definitions, different properties are difficult to compare for non-experts, making it hard to judge which scheme provides the best security-efficiency trade-off for a given application.

In this work, we challenge the approach of defining UE as a primitive with a set of properties. As an alternative, we propose to treat UE as an interactive protocol, whose goal is to implement secure outsourced storage, using limited and imperfect resources (such as a small, leakable memory). To facilitate this approach, we introduce a framework that allows to easily formalize different security guarantees and available resources, making security-efficiency trade-offs of UE protocols easy to compare.

We believe that our approach opens the way for many constructions of secure storage that are not compatible with the currently defined syntax of UE. Indeed, we propose two new protocols: one for the setting with adversaries who control randomness (an attack vector so far not considered for UE), and one for the setting with adversaries that actively tamper with memory. Both protocols provide stronger confidentiality guarantees than all existing UE schemes.

1 Introduction

1.1 Updatable Encryption

Key rotation and updatable encryption. In today's globalized world, the need for (private) data to be constantly available on many devices and to many (authorized) users is evident. A natural way to achieve this is outsourcing storage,

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

namely, uploading data to a commercial platform (often called a cloud). Unfortunately, this introduces new attack vectors, as the platform cannot be trusted to always protect confidentiality and integrity of data. To prevent these new attacks, data is encrypted and the secret key is distributed among authorized devices. However, this is still not fully satisfactory, since, first, devices may get compromised and reveal the key (e.g. if they are lost or catch a virus) and, second, a key should not be used too many times (e.g. due to cryptanalytic attacks). Mitigating these attacks is the goal of key rotation, which mandates periodically replacing the key under which (already stored) data is encrypted.

Efficiently implementing an outsourced storage service that supports key rotation is not an easy task. Let us first state the goal more precisely. For simplicity, from now on we consider only two parties: the host, i.e. the platform where data is uploaded, and the data owner, who represents all entities that can access the data.[3] They are both honest, but also careless — any piece of information they see or store, such as a key or a received message, may potentially leak to the adversary or be modified by her. The goal is to design a protocol for the parties that constructs a storage service for the owner, which protects confidentiality and integrity of as many stored messages as possible, given which pieces of information leaked or were corrupted.

The obvious approaches to achieve this are either insecure or inefficient. For example, if one uses standard symmetric encryption, it is not clear how to implement key rotation: sending the old and the new key to the host achieves suboptimal security, because information sent to the host is sufficient to decrypt all messages. Alternatively, the owner could download all data and re-encrypt it himself, but this is clearly not efficient.

Enabling more efficient protocols is the goal of updatable encryption (UE) introduced by Boneh et al.[BLMR13]. In essence, UE has the functionality of symmetric encryption, but offers two additional algorithms for key rotation: nxt (meant for the owner), given the old and the fresh key, generates a short *update token*, and upd (for the host), given a token and a ciphertext, updates it to the new key. Importantly, the token does not reveal any information about the keys.

Defining secure UE. UE has received considerable attention in the cryptographic community [BLMR13, EPRS17, LT18, KLR19, BDGJ20, Jia20, BEKS20, CLT20]. It comes in surprisingly many flavors and with many different security properties. First, there are multiple versions of syntax. For example, ciphertext-independent UE, where nxt generates a single token for all ciphertexts and ciphertext-dependent UE, where nxt generates one token per ciphertext and gets as input the ciphertext's short header (sent by the host). Independently, one can consider deterministic or randomized upd algorithm. Regarding security properties, there are notions analogous to IND-CPA, IND-RCCA and IND-CCA. The properties related to integrity include analogues of INT-PTXT and INT-CTXT.

[3] Considering a single owner, we implicitly assume that there is an external mechanism which synchronizes all devices. Achieving this is outside the scope of this work.

There are also properties formalizing secrecy of metadata, such as how many times a ciphertext has been updated.

Different syntaxes and security properties offer different trade-offs between security and efficiency, which are not always obvious to compare. This makes the task of choosing the right scheme for given application difficult. For instance, randomized upd is less efficient in terms of randomness and one cannot meaningfully define IND-CCA for it. However, it allows to achieve an additional property that an updated ciphertext looks like a fresh encryption even given the previous ciphertext and update token. As another example, notions that consider active attackers are defined slightly differently in different papers. Namely, sometimes the adversary can inject arbitrary ciphertexts only to the decryption algorithm (e.g. IND-CCA in [BDGJ20]), and sometimes to both decryption and upd (e.g. IND-RCCA in [KLR19]). Intuitively, the first version appears too weak, since it offers no guarantees if an active attacker modifies the host's storage and then the host applies upd. On the other hand, it is much more common in the UE literature and the difference is sometimes not made explicit.

1.2 Contributions

We claim that viewing UE as a primitive with a fixed syntax and a set of security properties is perhaps too restrictive. Instead, we propose a different approach, where we directly formalize the goal of UE. That is, UE is not a scheme but an interactive protocol between the owner and the host, the goal of which is to implement a secure storage service in the setting where different pieces of information may leak to the adversary or be modified by her.

Our modeling framework offers more flexibility than the standard approach and makes it easy to express different confidentiality and integrity guarantees. Moreover, it precisely formalizes resources required by protocols (such as channels and storage), making the security-efficiency trade-offs achieved by different constructions easy to see. Our framework generalizes all existing definitions, as well as enables finding new trade-offs, which would require UE with yet different syntax.

We then give three instantiations of our framework, focusing on ciphertext-independent UE. The first explains the known notion of IND-CPA security. In the second instantiation, we consider the setting where the adversary can control randomness, an attack vector so far not considered for UE. Not being constrained by any fixed syntax, we design a scheme with better bad-randomness resistance than what is possible with any existing syntax. Our construction is very simple and ready to implement in practice. Finally, the third instantiation considers active attackers who can modify the host's storage. While this attack vector is more realistic than, e.g., injecting messages on the channel (since the storage is long-lived), most existing security notions are not suitable, as they offer no guarantees if the host updates injected ciphertexts. Our construction protects confidentiality of data against active attackers. It builds on a suitable IND-RCCA secure UE scheme [LT18].

Framework for defining UE. In particular, we define security of UE protocols using the simulation-based paradigm, in a composable framework. This means that the real-world execution of a UE protocol should be indistinguishable from the ideal-world execution, where a trusted storage service keeps the owner's messages secret and all leaked information is simulated without them.

More precisely, the real world consists of a number of limited and imperfect resources: memories, channels and randomness sources (then the protocol must be deterministic and stateless). Each resource has bounded capacity and can only be used once, which formalizes efficiency. Moreover, the adversary may corrupt each resource and obtain read and/or write access to the contents (i.e., the stored, sent or sampled value). Finally, to express assumptions about the real world (e.g. authenticated channel), each resource is parameterized by two predicates: can-leak and can-inject, which control the adversary's read and write access, respectively. The leakability and injectability guarantees of a resource may change over time, for example, the first channel is leakable only once the second channel leaks. This is expressed using events [JMM19]: each resource can trigger events, such as leaked, and the global list of all events that happened so far is the input to can-leak and can-inject predicates.

The ideal world, on the other hand, consists of the secure storage service and a simulator. The ideal storage formalizes security of a UE protocol. It allows the owner to store, retrieve and delete messages, as well as to update all ciphertexts at once. Then, it gives to the simulator read and/or write access to only a subset of stored messages. Whether the access to a message is granted is determined by the can-leak and can-inject predicates, parameters of the storage. As in the real world, they can depend on the list of all events (such as the owner updating).

We highlight some relevant points of our model:

- *Flexible assumptions and guarantees.* Different assumptions and guarantees can be expressed by adjusting can-leak and can-inject predicates in the real and the ideal world, respectively.
- *Clear semantics.* The semantics of our statements are simple — one can think of the protocol as of the idealized trusted storage — making it easy to judge if a statement is suited for a given application. In contrast, for game-based definitions, judging if a given combination of security properties is exactly what is needed tends to be hard.
- *Flexible syntax and comparable efficiency.* The protocol can use its resources in an arbitrary way, and the assumed resources make it easy to compare efficiency. For instance, a protocol using ciphertext-dependent UE simply requires more channels, on which the host can send all ciphertext headers. In a different protocol, the host could collect information from all stored ciphertexts into a single shorter header. Such protocol would have efficiency between ciphertext-dependent and ciphertext-independent UE. (Its security guarantees would probably also be in between, since leaking the token would not affect messages stored after it is created.) This and many other examples show that hard-coding the syntax is unnecessarily restrictive.

– *Stronger statements.* The type of statements we make is that a protocol improves arbitrary guarantees of real-world resources. For example, a protocol using IND-CCA secure UE should improve confidentiality, while preserving integrity protection. This is stronger than a standard statement formalizing IND-CCA, where the real-world resources are always injectable.
– *Easy to extend.* While we focus on memory, channels and randomness, one can easily model different assumptions, such as a common reference string. This may enable more efficient solutions than standard UE.

Randomness corruption. As bad randomness generators are a very realistic attack vector (a well-known example is the backdoored DualEC PRNG [SS06, SF07]), it comes as a surprise that they have never been considered in the context of UE. In fact, existing schemes achieve very different guarantees in the presence of attackers who can corrupt randomness. For example, consider leaking randomness used to encrypt a message. With the RISE scheme [LT18] (where encryption is essentially ElGamal) this allows to decrypt the message, while the BLMR [BLMR13] and SHINE [BDGJ20] schemes remain secure. Second, consider leaking randomness used by nxt to generate a token. The way SHINE is defined, nxt first chooses the new key at random, and then computes the token using both keys. This means that leaked randomness contains the new key and can be used to decrypt all ciphertexts. An equivalent way to define SHINE is to generate the token at random and compute the new key. Intuitively, this is more secure, since the token cannot be used to decrypt.

In this work, we propose a construction that improves upon all existing schemes. Our protocol is for the setting where randomness may be chosen by the adversary. Roughly, the first idea is to replace the randomized encryption algorithm with one that is nonce-based and hence deterministic. This is done in a generic way, using a PRF, as in [Rog04]. This is clearly better, since nonces do not need to be secret. To choose the nonce, our protocol assumes that each stored message has a unique identifier. This assumption seems reasonable in practice — if the owner can keep a key, then he can also store a message counter.[4]

The only randomized operation left is token generation nxt. The second idea is to minimize the number of these operations that can be affected by adversary's randomness. In particular, the randomness for nxt is derived from a fresh random value mixed with the old key. This means that randomness is secret as long as either the old key or the fresh randomness is secret.

We build our protocol from (slightly modified) SHINE0 [BDGJ20] and prove that it achieves desired security in the ideal-cipher model, assuming DDH. We stress that it is not possible to generically construct a protocol for our setting from a UE scheme with some property, because all existing properties deem secure protocols that trivially break if the adversary controls randomness. While we could introduce a new syntax for nonce-based UE, as well as security games that consider randomness corruptions, this would be complicated and is not necessary with our framework.

[4] If he has multiple devices, then each device can have an identifier and a local counter.

Injections. Assuming that the host can protect integrity of stored data can often be considered unrealistic, as storing large amounts of data for long periods of time increases the risk of data corruption (e.g. due to system errors, or hackers). Therefore, we now consider active attackers who can, in addition to leaking, inject arbitrary values into the host's memory. The goal is to improve confidentiality, while preserving any (very weak) authenticity guarantees of real-world memory. Intuitively, this means that the attacker can inject, but only values unrelated to encrypted ones.[5]

To achieve this, first, we use the UE scheme of [LT18], which provides IND-RCCA security, where the adversary is allowed to update arbitrary ciphertexts. The IND-RCCA notion is similar to the well-known IND-CCA, but it does not consider coming up with a different ciphertext of the exact same message an attack. This weaker notion is sufficient for our goals. Second, we note that the ideal storage should be considered as a whole, so, naturally, the expected behavior is that if the owner retrieves a message he stored under identifier i, then he does not receive the message he stored under j. Since UE schemes are oblivious to identifiers, we encrypt the message with its identifier. A message with wrong identifier is treated as malformed.

1.3 Related Work

Ciphertext-dependent and independent UE. There are two main flavors of UE: ciphertext-dependent [BLMR13, EPRS17, BEKS20, CLT20], where the update token depends on the ciphertext, and ciphertext-independent [LT18, KLR19, BDGJ20, Jia20], where one token can be used to update any ciphertext.

Ciphertext-independent UE is more efficient, because a full key rotation requires sending only one token to the host. In contrast, in ciphertext-dependent UE, the host sends a header for each ciphertext and then the owner sends a token for each header (which means that the communication cost scales linearly in the number of stored messages). On the other hand, ciphertext-dependent schemes have better security guarantees in the presence of fully active attackers. In particular, the only scheme with the strongest confidentiality (IND-CCA) and integrity protection (INT-CTXT) in this setting [CLT20] is ciphertext-dependent. The only ciphertext-independent scheme for this setting [KLR19] achieves weaker guarantees (IND-RCCA and INT-PTXT).

Active attackers. Unlike standard encryption schemes, UE schemes have two algorithms that take as input ciphertexts, and hence can be subject to injections: decryption dec and update upd. Fully active attackers, who can inject to both algorithms, are only considered in [KLR19, CLT20]. Most notions formalizing

[5] One may notice that this setting does not make much sense for standard symmetric encryption, where one provides authenticity before considering confidentiality. This is not the case in the presence of key leakage, which makes some injections inherent. Such injections should not violate confidentiality, which requires a stronger notion than, e.g., IND-CPA.

integrity consider 'semi-passive" attackers, who can inject to dec but not upd [EPRS17, BEKS20, BDGJ20]. There are also notions that consider adversaries who inject into upd but not dec, e.g. IND-CPA in [BLMR13].

Hiding metadata. In addition to traditional confidentiality and integrity protection, the literature defines properties that formalize confidentiality of various metadata, for example, how many times (if at all) a ciphertext has been updated [BDGJ20, BEKS20]. Another property formalizes a form of unlinkability — after two ciphertexts are updated, it should be hard to tell which is which. See [BDGJ20] for a better overview of such properties.

1.4 Outline of the Paper

Section 2 presents background on the constructive cryptography framework and updatable encryption, necessary to define our framework in Section 3. Section 4 presents the first instantiation of our model, explaining IND-CPA secure UE. It serves as a basis for the next two instantiations, considering randomness corruption in Section 5 and injections Section 6. Section 7 contains conclusions and open problems.

2 Preliminaries

2.1 Notation

We use $y \leftarrow\!\$\ \mathsf{alg}(x)$ to denote assigning to the variable y the output of a randomized algorithm alg on input x. For simplicity, we make the security parameter implicit. Furthermore, we use the **req** command, which is shorthand for verifying a condition and returning $\perp$ if the condition is false.

2.2 Updatable Encryption

The best way to explain UE is to describe how the scheme will be used. See Fig. 1 for an illustration. The execution proceeds in epochs, where each epoch corresponds to one secret key and epoch change corresponds to key rotation. The first epoch $e = 0$ is created by the owner generating the first key $k_0 \leftarrow\!\$\ \mathsf{ue.kg}()$. He can now use k_0 to encrypt and decrypt messages, just like in a standard encryption scheme. To create a new epoch, the owner takes the current key k_e and runs $(k_{e+1}, \Delta_e) \leftarrow\!\$\ \mathsf{ue.nxt}(k_e)$ to generate the next epoch's key k_{e+1} and the update token Δ_e. Then, he sends Δ_e to the host, who replaces each stored ciphertext c_e by $c_{e+1} \leftarrow\!\$\ \mathsf{ue.upd}(\Delta_e, c_e)$. The new ciphertext c_{e+1} can be decrypted using k_{e+1} (but not k_e).

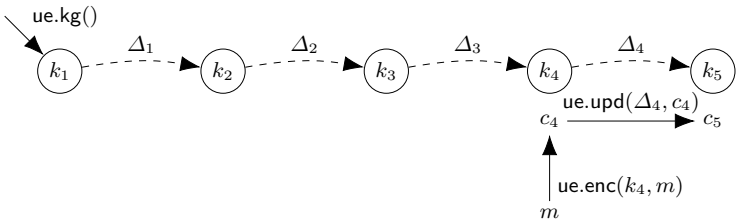

Fig. 1: An illustration of a UE scheme being used. Nodes represent epochs and edges (dashed arrows) represent key rotations. In this scenario, a message m was encrypted during epoch 4 under k_4, resulting in a ciphertext c_4. When epoch 5 was created, c_4 was updated using Δ_4, resulting in c_5, encryption of m under k_5.

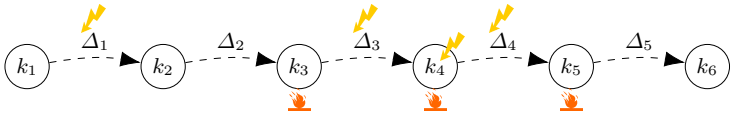

Fig. 2: An illustration of corruptions of keys and tokens. If the keys and tokens marked by ⚡ are corrupted, then all epochs marked by 🔥 are exposed (i.e., those reachable from a corrupt node via corrupt edges, irrespective of direction).

Syntax and correctness. We adopt the syntax of [LT18]. A UE scheme is defined over a message space $\mathcal{M}$, ciphertext space $\mathcal{C}$, key space $\mathcal{K}$, and token space $\mathcal{T}$, and consists of the following algorithms:

- $k_0 \leftarrow\!\!\$\ \mathsf{ue.kg}()$ generates the first secret key $k_0 \in \mathcal{K}$
- $c_e \leftarrow\!\!\$\ \mathsf{ue.enc}(k_e, m)$ given a key $k_e \in \mathcal{K}$ and a message $m \in \mathcal{M}$, outputs a ciphertext $c_e \in \mathcal{C}$
- $m \leftarrow \mathsf{ue.dec}(k_e, c_e)$ given a key k_e and a ciphertext $c_e \in \mathcal{C}$, outputs a message $m \in \mathcal{M}$ or $\perp$
- $(k_{e+1}, \Delta_e) \leftarrow\!\!\$\ \mathsf{ue.nxt}(k_e)$ given a key $k_e \in \mathcal{K}$, outputs a fresh key $k_{e+1} \in \mathcal{K}$ and an update token $\Delta_e \in \mathcal{T}$
- $c_{e+1} \leftarrow\!\!\$\ \mathsf{ue.upd}(\Delta_e, c_e)$ given a token $\Delta_e \in \mathcal{T}$ and a ciphertext $c_e \in \mathcal{C}$, outputs updated ciphertext $c_{e+1} \in \mathcal{C}$

Correctness, intuitively, requires that any epoch-e ciphertext can be decrypted using epoch-e key. We refer to [LT18] for a formal definition.

Confidentiality notions. In this paper, we use two confidentiality notions for UE: `IND-ENC-CPA`, where we adopt the definition of [BDGJ20], and `IND-ENC-RCCA` defined in [KLR19]. (ENC in `IND-ENC-CPA` indicates that we only consider secrecy of the message, and not of metadata.) We outline them in the rest of this section. See Appendix A.2 for precise definitions.

Intuitively, in IND-ENC-CPA the challenger allows the adversary to drive a scenario envisioned for UE. In particular, the adversary can create a new epoch, encrypt messages under the current key, and update a ciphertext to the current epoch. Any generated or updated ciphertext is given to the adversary, while keys and tokens are secret and can only be obtained from a corrupt oracle. At some point the adversary requests a standard IND-CPA challenge and continues driving the execution. Finally, the adversary wins if it guesses the challenger's bit and it does not trivially win, i.e., the challenge ciphertext cannot be trivially decrypted computed using the keys and tokens to the adversary. See Fig. 2 for an illustration.

IND-ENC-RCCA security is a relaxed variant of CCA, where the adversary is allowed to query the decryption oracle with any arbitrary ciphertext, except when it decrypts to either of the two challenge messages. Moreover, the adversary is allowed to query the Upd oracle for arbitrary ciphertexts, i.e., these need not be first generated using the encryption oracle.

2.3 Constructive Cryptography

We state our results in the constructive cryptography (CC) framework [MR11, Mau11]. We follow the presentation in [JMM19].

Resources. An important notion in CC is that of a resource (resources are similar to functionalities in the UC framework [Can01]). A resource is simply a reactive system that can be accessed via one or more *interfaces*. After receiving input at an interface, the resource immediately returns outputs at the same interface. CC does not define any computational model and is only concerned with input-output behavior of resources. We typically describe this behavior using pseudo-code and resources should be thought of as black boxes with behavior specified by the code. (Formally, resources are modeled as random systems [Mau02], where the interface address is encoded as part of the inputs.)

We usually distinguish between interfaces meant for parties, e.g. an interface for data owner called O, and those meant for the adversary, typically called E (for Eve). For example, consider an insecure memory resource defined in Fig. 3, which allows the owner to store a message and retrieve the last stored value. The owner's message is available at the adversary's interface, modeling that the memory is insecure. Note that resources can have an initialization procedure initializing any and all global variables (all other variables local), which runs the first time any input is triggered on any interface.

Converters. A protocol is modelled by converter-systems, each representing all local computations performed by a party. Such systems have two types of interfaces: the inner interface that connect to (an interface of) different resources, such as memories, and the outer interface that is exposed to the environment. Upon receiving an input on its outside interface, the converter can trigger (a bounded number of) inputs to the resource via its inner interface, perform some

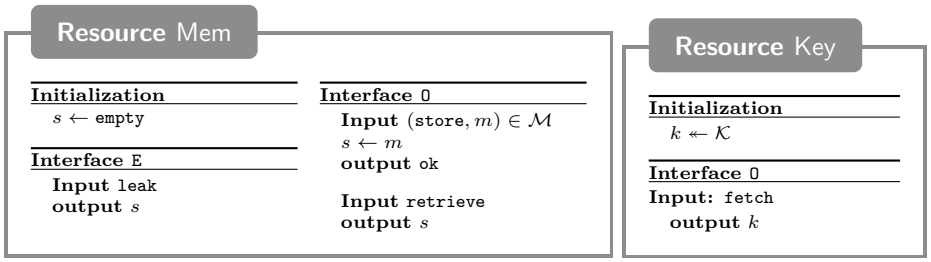

Fig. 3: The memory and key resources the 0 and E interfaces is are understood to be controlled by the (honest) data owner and the adversary, respectively.

internal computations, and eventually output some value on the outside interface. Similar to resources, we describe converters using pseudo-code.

For example, consider a protocol in which the owner uses a secret key to encrypt data in an insecure memory. This is represented by a converter prot which, on the inner interface, connects to the memory and key resources defined in Fig. 3 and on the outer interface exposes the same interface as the memory resource. The behavior of prot can be described as follows: on input $(\texttt{store}, m)$ on the outer interface, fetch the key from the key resource (using the inner interface), store m encrypted under the fetched key in the memory resource and output ok on the outer interface.

A converter prot whose inner interfaces is connected to an interface I or a resource R yields a new resource, denoted $\mathsf{prot}^I\mathsf{R}$ (hence the name — a protocol converts R into a new resource). In the example above, the new resource is $\mathsf{prot}^0[\mathsf{Mem}, \mathsf{Key}]$ (denoting that prot is connected to all 0 interfaces).

Global Events. We use the enhanced version of constructive cryptography with the notion of globally observable *events*, introduced in [JMM19], which encapsulates the dependencies between resources. That is, resources can trigger events that get added to a *global event history* (GEH), a global (ordered) record of all events that have been triggered in the environment, that is accessible to all resources. So, resources use the global event history in order to get a sense of the state of other resources, and act accordingly. Formally, events are a generalization of monotone binary outputs (MBO) introduced by Maurer et al. [MPR07]. Roughly, an MBO is a value that can change from 0 to 1 but not back, which can be interpreted as a single event happening once the MBO changes to 1. An event then just corresponds to a named MBO and the global event history (GEH) $\mathcal{H}$ is a list of event names.

In this work, the global event history is represented by $\mathcal{H}$. Further, to append the event n to the GEH we use the notation $\mathcal{H} \xleftarrow{+} \mathcal{E}^{\mathsf{n}}$, and to check if event n_1 happened before event n_2 (i.e., $\mathcal{E}^{\mathsf{n}_1}$ got appended to the GEH first) we use the notation $\mathcal{E}^{\mathsf{n}_1} \prec \mathcal{E}^{\mathsf{n}_2}$. Lastly, we use the notation $\mathcal{E}^{\mathsf{n}}$, which is shorthand for $\mathcal{E}^{\mathsf{n}} \in \mathcal{H}$. Importantly, note that the GEH from the real-world and the GEH from the ideal world are completely independent, and resources in one do not have access to

Fig. 4: Execution of the protocol in the real world (left) and the ideal world with the simulator attached to Eve's interface (right).

the GEH of the other. This is due to the fact that these are two separate and independent execution environments.

Contrary to the treatment in [JMM19], we allow the same event to be added to the GEH multiple times. When this happens, however, the event gets appended with a counter, indicating which instance of the event this is. For example, the first time $\mathcal{H} \xleftarrow{+} \mathcal{E}^{\mathsf{n}}$ is called, the event $\mathcal{E}^{n_1:0}$ gets appended to the GEH. Then, whenever $\mathcal{H} \xleftarrow{+} \mathcal{E}^{\mathsf{n}}$ is called again, $\mathcal{E}^{n_1:1}$ gets appended to the GEH, and so on. If we think of the GEH as an array, adding an event corresponds to scanning the array for events with the same label and resource id, maintaining a counter for the number of these, and appending the new event with the final tally. Then, we can reference an instance of an event by specifying the index of it. Further, if no index is specified, this implicitly translates to the 0-th index. So, $\mathcal{E}^{n_1:i}$ would correspond to the i-th instance of the event, and $\mathcal{E}^{\mathsf{n}}$ would correspond to the event's first (and potentially only) appearance in the GEH.

We associate events with a label (e.g., leaked), and an id representing the resource that triggered it. We then use the notation $\mathcal{E}_{id}^{\mathsf{label}}$ to fully identify an event.

Resources and converters have read/write access to the global event history. 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.

The Construction Notion. The goal of a protocol is to convert the so-called real-world resources into a better ideal-world resource; that is, to *construct* the ideal resource. If this is the case, then instead of the protocol with its real-world resources, one can think of the ideal resource, which is secure by design. For instance, the protocol prot discussed above constructs a secure memory, which leaks to the adversary only message length, from an insecure memory and a key.

To formalize this, we first introduce the simulator, which is a converter attached to the adversary's interface of the ideal resource. Its task is to emulate the adversary interfaces of the real world resources. Now the real world with the assumed resources R and the protocol and the ideal world with the ideal resource S and a simulator should be indistinguishable. See Fig. 4 for an illustration. Intuitively, if this is the case, then whatever the adversary could compute using

her interfaces of the real-world resources, she could also compute using the ideal resource. Hence, the real world is just as good.

We proceed to formalize indistinguishability of two systems R' and S'. To this end, we consider a *distinguisher* $\mathcal{D}$ — a system with access to all of the resource's interfaces, whose goal is to determine which system it is interacting with.

The distinguisher also has read/write access to the global event history $\mathcal{H}$, which we denote $\mathcal{D}^{\mathcal{H}}$. Thus, the real and the ideal world can only be indistinguishable if they trigger analogous events. That is, after $\mathcal{D}$'s interaction with either world, the sequence of events must be the same. Since resources in both worlds are different, however, we cannot have exactly equal events. Instead, we allow renaming of events, by defining a function τ that maps ideal-world events to their real-world analogues. This means that in the ideal world $\mathcal{D}$ interacts with $\mathcal{H}$, denoted $\mathcal{D}^{\mathcal{H}}$, and in the real world $\mathcal{D}$ interacts with $\tau(\mathcal{H})$, denoted $\mathcal{D}^{\tau(\mathcal{H})}$

Definition 1. *We say that two systems* R' *and* S' *are indistinguishable under renaming* τ*, denoted*

$$\mathsf{R}' \mathrel{\hat{\approx}} \mathsf{S}'$$

if there exists a relabeling τ *such that* τ *only renames events triggered by* S'*, and for all efficient event-aware distinguishers* $\mathbf{D}^{\mathcal{H}}$*, the advantage* $\Delta^{\mathbf{D}^{\mathcal{H}}}(\mathsf{R}', \mathsf{S}') \coloneqq \Pr\left[\mathbf{D}^{\tau(\mathcal{H})}(\mathsf{S}') = 1\right] - \Pr\left[\mathbf{D}^{\mathcal{H}}(\mathsf{R}') = 1\right]$ *is negligible.*

3 Modeling Guarantees of Updatable Encryption

Recall that the goal of an updatable encryption scheme is to enable constructing a storage service for the data owner, maintained with the help of the host. Moreover, this should be achieved using imperfect and limited resources. In this section, we formalize this goal using constructive cryptography. The model consists of two parts. First, we formalize the imperfect real-world resources available to protocols, such as channels and memories that are leakable or even injectable, and have limited capacity. Then, we define the resource $\mathsf{Storage}$, which models an ideal storage service. The goal of a protocol is then to construct $\mathsf{Storage}$ using a limited number of real-world resources.

3.1 The Real-World Resources

The resources we are concerned with are memories for the owner and the host, communication channels between them and randomness. The corresponding memory, channel and randomness systems are defined in Fig. 5.

At a high level, the memory allows a user at interface $\mathtt{U}$ to store a single message and later delete it, the channel allows a sender at interface $\mathtt{S}$ to send a single message to a receiver at interface $\mathtt{R}$, and the randomness source allows a user to fetch a single random value. The user, sender and receiver interfaces are parameters of the resources and can be set, e.g., to the owner $\mathtt{O}$ or the host $\mathtt{H}$. Moreover, each resource has a parameter id, identifying this particular

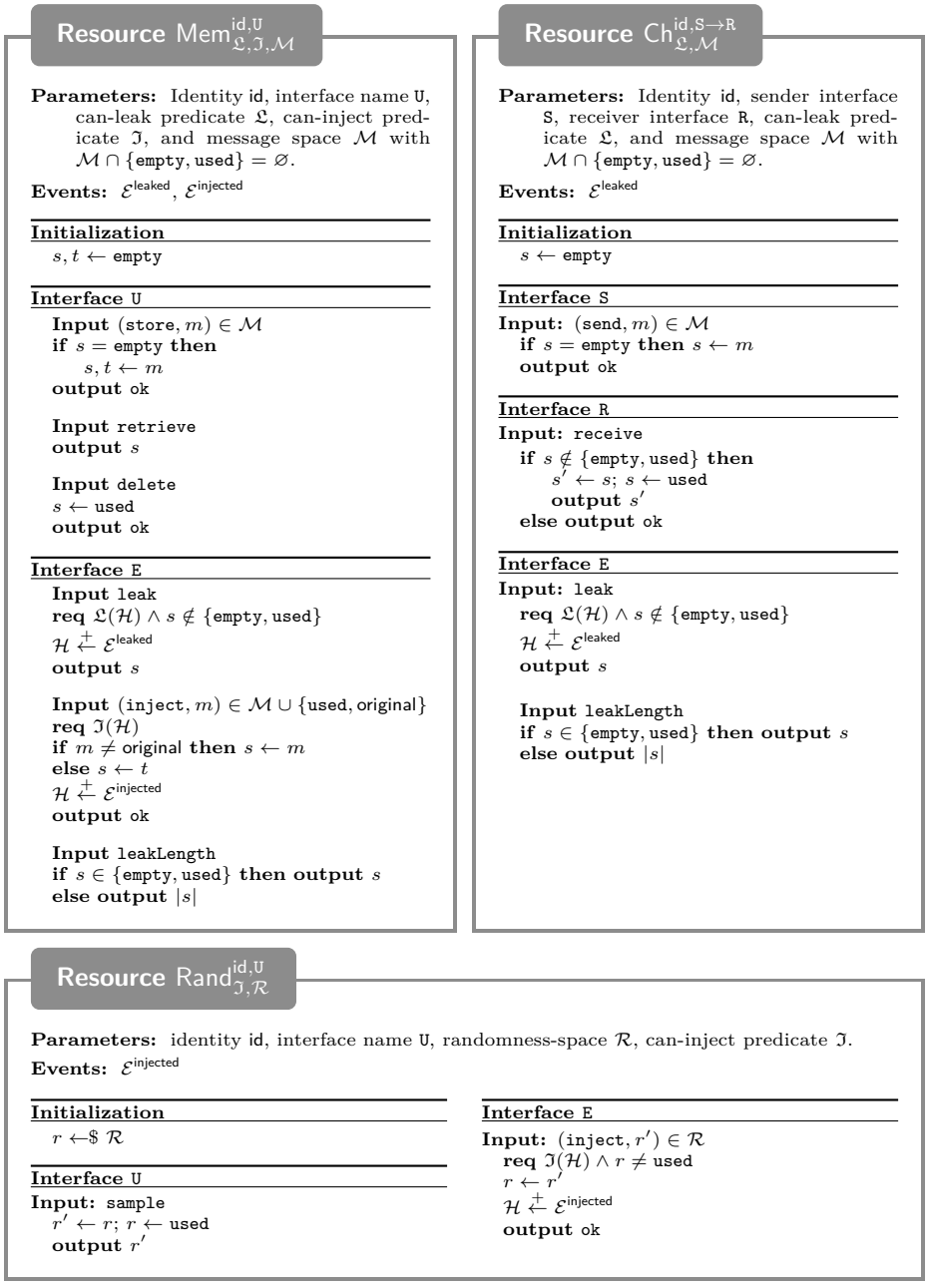

Fig. 5: The real-world resources: leakable and injectable memory, leakable channel and injectable randomness.

instance (for example, a protocol can use the i-th channel, meaning it uses a channel with $\mathsf{id} = i$). This basic functionality is quite simple. We now explain how further details of the resources formalize efficiency and imperfectness of the real world.

Formalizing efficiency. To model limited storage and communication bandwidth, the memory and channel resources are each parameterized by a message space $\mathcal{M}$ and only a value from $\mathcal{M}$ can be stored or sent. Moreover, to model limited number of communication rounds, the channels are single-use (as a result, protocols using more rounds require more channels). Finally, to model limited amount of randomness, the resource outputs only one value sampled from a space $\mathcal{R}$.

Formalizing imperfectness. The goal is to reflect the setting where any piece of information may (independently) leak or become corrupted (due to the host's or the owner's negligence). To this end, we make resources leakable and injectable, meaning that the adversary has read and write access to each stored, sent or sampled value (via inputs leak and inject). For simplicity, the randomness resource is only injectable (here, injecting gives strictly more power than leaking) and the channel is only leakable (we do not consider this attack in this work, so we do not define it; see also Section 7).

Each time the adversary leaks or injects a value, which should be thought of as a corruption, a leaked or injected event is triggered. Looking ahead, the guarantees provided by protocols will depend on a particular sequence of such events. Note that a resource can be corrupted multiple times.

Furthermore, the resources are parameterized by predicates can-leak $\mathfrak{L}$ and can-inject $\mathfrak{I}$, which control the adversary's read and write access, respectively. The input to the predicates is the event history $\mathcal{H}$, modeling that the access rights change over time. For example, a memory with $\mathfrak{I}$ always false offers integrity protection (which is an assumption about the real world). Another example is a memory with $\mathfrak{I}$ that is initially false and becomes true after a number of store operations.

Finally, we note that leakability and injectability are orthogonal, e.g. the adversary may inject a value into a memory without knowing the value she overwrites. Therefore, it makes sense to also give her the possibility to undo the injection and "inject" the original value (this is something she cannot do if can-leak is false).

Implicit and explicit resources. In principle, in constructive cryptography, everything that matters, for example storage, randomness and even computation, is modeled explicitly as a resource, while protocols only describe a way to connect resources they have available. However, for simplicity, one often does not make resources that are not of concern for a given statement explicit and describes protocols that use them implicitly without limitations.

For example, we model explicit memory resources and protocols must be stateless. Randomness is a concern only for some of our statements, where we

use explicit randomness resources and the protocols must be deterministic. On the other hand, in all other statements, we describe randomized protocols.

3.2 The Ideal Storage Service

This section defines the ideal storage service Storage, constructed by protocols using updatable encryption. At a high level, Storage contains n memory cells, and offers additional functionality related to being maintained by the owner and the host. A memory cell is represented as a memory resource defined in Fig. 5 (formally, cells are not individual resources, but sub-modules of a single storage resource). The can-leak and can-inject predicates of a cell formalize, in a simple way, the achieved security guarantees for a single value stored by the owner.

The storage service Storage is defined in Fig. 6. In essence, it allows the owner to access different cells, as well as to update all cells. Before we describe Storage in more detail, we note that all "inputs" of the owner discussed so far (such as store or update) are in fact "actions", which will be implemented in the real world by interactive protocols between the owner and the host. For example, retrieving a value requires the owner to request it, then the host to send the ciphertext, and only then can the owner decrypt it and output the value. Technically, this means that in the real world each action requires a sequence of activations of the owner's and the host's protocols, in order to complete. This needs to be reflected in the ideal world, where such activations correspond to additional inputs to the ideal resource Storage.

For the above reason, we distinguish two types of inputs at the owner interface of Storage: first, those that initiate an action, such as `store` or `update`, and, second, an additional input `activate`. Further, the resource has a host interface that takes only the input `activate`. Each action has a sequence of `activate` inputs from the owner and the host that need to follow it. The last such input in the sequence always comes from the owner and it returns the output of the action, e.g. the retrieved value. For simplicity, we make actions atomic — after an action is initiated, Storage blocks (i.e., outputs $\perp$) until the sequence of activations is completed. That is, no new actions can be initiated during that time. We stress that this only affects the owner and the host, while the adversary can interact with the resource at any time.[6]

For flexibility (different protocols may require different sequences), we parameterize the resource by a dictionary `ActSeq` that maps actions, represented by commands initiating them, such as `store`, to activation sequences. An activation sequence is represented by a stack of values `owner-activate` and `host-activate`, where the activations should be executed from the one on the top of a sequence to the one on the bottom.

[6] One can also consider a weaker statement, where no guarantees are given unless the correct sequence is executed after each input (formally, this means restricting the class of distinguishers to those that input `activate` correctly). We do not formalize this, since our protocols achieve the stronger guarantee. However, deriving the weaker notion from our definition is straightforward.

Resource Storage$_{n,\vec{\mathfrak{L}},\vec{\mathfrak{I}},\mathcal{M},\texttt{ActSeq}}$

Parameters:
- Maximum number n of stored values,
- Can-leak predicates $\vec{\mathfrak{L}} = (\mathfrak{L}^1, \ldots, \mathfrak{L}^n)$ and can-inject predicates $\vec{\mathfrak{I}} = (\mathfrak{I}^1, \ldots, \mathfrak{I}^n)$,
- Message space $\mathcal{M}$,
- Dictionary `ActSeq`, assigning to each action `act` $\in \{$`store`, `retrieve`, `delete`, `update`$\}$ a sequence of activations that need to follow it, represented as a stack.

Variables:
- Memory cells $\mathsf{Mem}^{1,0}, \ldots, \mathsf{Mem}^{n,0}$ with can-leak predicates $\mathfrak{L}^1, \ldots, \mathfrak{L}^n$, can-inject predicates $\mathfrak{I}^1, \ldots, \mathfrak{I}^n$ and message space $\mathcal{M}$,
- Values `curAct`, `curCel`, `curRet`, `curSeq`, each initialized to `none`.

Interface `0`

Input $(\texttt{inp}, i) \in (\{\texttt{retrieve}, \texttt{delete}\} \cup \{\texttt{store}\} \times \mathcal{M}) \times \mathbb{N}$
req `curAct` $=$ `none`
call `ret` $\leftarrow$ `inp` at int. `0` of $\mathsf{Mem}^{i,0}$
if `inp` $= (\texttt{store}, m)$ **then** `curAct` $\leftarrow$ `store`
else `curAct` $\leftarrow$ `inp`
$(\texttt{curCel}, \texttt{curRet}) \leftarrow (i, \texttt{ret})$
`curSeq` $\leftarrow$ `ActSeq[curAct].copy()`
$\mathcal{H} \xleftarrow{+} \mathcal{E}^{\texttt{start},\texttt{curAct},\texttt{curCel}}$
output `ok`

Input `update`
req `curAct` $=$ `none`
$(\texttt{curAct}, \texttt{curCel}, \texttt{curRet}) \leftarrow (\texttt{update}, 0, \texttt{ok})$
`curSeq` $\leftarrow$ `ActSeq[update].copy()`
$\mathcal{H} \xleftarrow{+} \mathcal{E}^{\texttt{start},\texttt{curAct},\texttt{curCel}}$
output `ok`

Input `activate`
req `curSeq.peek()` $=$ `owner-activate` $\wedge$ `curAct` $\neq$ `none`
`curSeq.pop()`
$\mathcal{H} \xleftarrow{+} \mathcal{E}^{\texttt{owner-activate}}$
if `curSeq.empty()` **then**
  $\mathcal{H} \xleftarrow{+} \mathcal{E}^{\texttt{curAct},\texttt{curCel}}$
  `curAct` $\leftarrow$ `none`
  output `curRet`
else output `ok`

Interface `E`

Input $(\texttt{command}, i)$
call `ret` $\leftarrow$ `command` at int. `E` of $\mathsf{Mem}^{i,0}$
output `ret`

Interface `H`

Input `activate`
req `curSeq.peek()` $=$ `host-activate` $\wedge$ `curAct` $\neq$ `none`
`curSeq.pop()`
$\mathcal{H} \xleftarrow{+} \mathcal{E}^{\texttt{host-activate}}$
output `ok`

Fig. 6: The resource representing the storage system with n memory cells.

In more detail, Storage works as follows. Information about the currently executed action is stored in the variables `curAct`, `curCel`, `curRet` and `curSeq`. In particular, `curAct` is the action's name (e.g. `store`), `curCel` is the memory cell the action is related to (an integer), and `curRet` is the output generated by the action (to be outputted upon the last activation). Finally, `curSeq` is the sequence of activations that still need to be executed before the action completes.

When no action is being executed, `curAct` is set to `none`. At this point, and only then, the owner can initiate an action by inputting $(\texttt{inp}, i)$, where `inp` is the input to a cell resource, e.g. $(\texttt{store}, m)$ and i is the index of the cell. Upon such input, Storage first inputs `inp` to the i-th Mem resource. If the output from Mem is not `empty` nor `used` (else, no action is initiated and Storage immediately outputs $\perp$), it is stored in `curRet` and the variables `curAct`, `curCel` and `curSeq` are initialized according to the input. Finally, it triggers an event signalizing

the action's start. The value `inp` can also be set to `update`, in which case i is ignored. The only consequence of an update is triggering an event, on which the can-leak predicate of the cells may depend.

While an action is being executed, the only inputs from the owner and the host accepted by Storage are `activate`. If the input is the next one in `curSeq`, it is removed from the stack and the activation event is triggered. Upon the owner's `activate`, the resource additionally checks if the action is complete, i.e., if `curSeq` is empty. If this is the case, an event denoting action's completion is triggered and the resource outputs the stored return value `curRet`.

4 Explaining IND-CPA

In this section, we show how an updatable encryption scheme can be used in a protocol that constructs the storage resource Storage from a number of leakable memories for the host and the owner, and a number of channels between them. In this construction, we do not consider randomness corruption or cost, and hence for simplicity we allow the protocol to be randomized.

4.1 Protocol

The protocol requires a UE scheme ue. It implements each action by a two-move protocol: 1) The owner sends to the host the action's name and the cell index on a "command" channel. For some actions, he sends more data on an additional channel. 2) The host performs the requested action and sends back data needed to compute the action's output, or an acknowledgment for actions without output. Recall that Storage requires that actions are atomic. To achieve this, the owner's protocol, after Step 1), remembers the current action and outputs $\perp$ until it receives an acknowledgment. Specifically, the actions proceed as follows:

Action store message m. The owner reads the current key k from his memory and sends $c \leftarrow\$ \mathsf{ue.upd}(k, m)$ to the host on an additional channel. The host stores c in the memory specified by the index.

Action update. The owner reads the current key k from his memory, computes $(k', \Delta) \leftarrow\$ \mathsf{ue.nxt}(k)$, stores k' in the next memory, erases k, and sends Δ on an additional channel. The host, for each ciphertext c, computes $c' \leftarrow\$ \mathsf{ue.upd}(\Delta, c)$, stores c' in the next memory and erases c.

Action retrieve. After receiving the command, the host sends the given ciphertext on an additional channel.

Action delete. After receiving the command, the host deletes the ciphertext.

To summarize, the protocol requires the real-world resources $\mathsf{R}^{\mathtt{UE}}$ defined in Fig. 7. Note that all memories are non-malleable. The protocol converters `ue-owner` and `ue-host` are formally described in Appendix B.1, where we take care of all the necessary bookkeeping.

$\mathsf{Mem}_{\mathcal{K}}^{(\mathsf{key},e),\mathtt{O}}$, $e \in [n_u]$	small memories, where the owner stores secret keys
$\mathsf{Mem}_{\mathcal{H}}^{(\mathsf{act},a),\mathtt{O}}$, $a \in [n_{tot}]$	small memories, which the owner uses to keep track of which action is being executed, if any.
$\mathsf{Mem}_{\mathcal{C}}^{(i,e),\mathtt{H}}$, $i \in [n_s]$	large memories, where the host stores ciphertexts
$\mathsf{Ch}_{\mathcal{C}}^{(\mathsf{ctx},i),\mathtt{O}\to\mathtt{H}}$, $i \in [n_s]$	channels, where the owner sends ciphertexts to store
$\mathsf{Ch}_{\mathcal{T}}^{(\mathsf{tok},e),\mathtt{O}\to\mathtt{H}}$, $e \in [n_u]$	channels, where the owner sends tokens
$\mathsf{Ch}_{\mathcal{C}}^{(\mathsf{ctx},i,j),\mathtt{H}\to\mathtt{O}}$, $i \in [n_s]$, $j \in [n_{sr}]$	a number of channels, where the owner retrieves ciphertexts: i-th ciphertext can be retrieved on channels $(\mathsf{ctx}, i, 1)$ to $(\mathsf{ctx}, i, n_{sr})$
$\mathsf{Ch}_{n_s\times\{s,r,d,u\}}^{(\mathsf{cmd},p),\mathtt{O}\to\mathtt{H}}$, $p \in [n_{tot}]$	channels where the owner sends commands he wants the host to execute
$\mathsf{Ch}_{\{1\}}^{(\mathsf{ack},p),\mathtt{H}\to\mathtt{O}}$ for $p \in [n_{tot}]$	channels where the host acknowledges executing commands

Fig. 7: The real-world resources $\mathsf{R}^{\mathtt{UE}}$, where n_s, n_d and n_u denote, respectively, the (desired) numbers of store, delete and update actions, n_{sr} denotes the number of retrieve actions *per stored value*, and $n_{tot} = n_s + n_d + n_u + n_s n_{sr}$. All can-leak predicates are arbitrary (except for the command and ACK channels, where $\mathfrak{L}$ is true) and all can-inject predicates are false.

4.2 Construction Statement

The protocol constructs the ideal storage resource $\mathsf{Storage}_{n_s,\vec{\mathfrak{L}},\vec{\mathfrak{I}},\mathcal{M},\mathtt{ActSeq}}$, where $\mathcal{M}$ is the message space of the UE scheme, all can-inject predicates in $\vec{\mathfrak{I}}$ are false and $\mathtt{ActSeq} \coloneqq \{\mathtt{act} : [\mathtt{owner\text{-}activate}, \mathtt{host\text{-}activate}]\ \mathtt{for\ act} \in \{\mathtt{store}, \mathtt{retrieve}, \mathtt{delete}, \mathtt{update}\}\}$. We need this specific activation dictionary since each action follows the same structure: the owner sends a message to the host, the host replies accordingly (an activation), and the owner receives this response (a second activation).

We proceed to define the can-leak predicate of the i-th cell, $\mathfrak{L}^i$. We first introduce some notation. Recall that the $\mathcal{E}^{\mathsf{updated}}$ events partition $\mathcal{H}$ into segments, which we call epochs (where each epoch corresponds to a single encryption key). We define the function $\mathsf{epoch}(\mathcal{H}, \mathcal{E}^{\mathsf{id}})$, which returns the set of all epochs in $\mathcal{H}$ that contain the event $\mathcal{E}^{\mathsf{id}}$, as follows:

$$\mathsf{epoch}(\mathcal{H}, \mathcal{E}^{\mathsf{id}}) \coloneqq e \mid (\mathcal{E}^{\mathtt{updated}:e} \prec \mathcal{E}^{\mathsf{id}} \wedge \neg\exists e' : \mathcal{E}^{\mathtt{updated}:e} \prec \mathcal{E}^{\mathtt{updated}:e'} \prec \mathcal{E}^{\mathsf{id}}) \\ \vee\ (e = 0 \wedge \mathcal{E}^{\mathsf{id}} \prec \mathcal{E}^{\mathtt{updated}:1})$$

Next, we define the predicate $\mathsf{exposed}(\mathcal{H}, e)$, which determines if the e-th epoch in $\mathcal{H}$ is exposed, that is, if the information leaked to the adversary is sufficient to decrypt ciphertexts from epoch e. Intuitively, e is exposed if there exists another epoch e' (before or after e) such that the owner's key in e', as well as all tokens

between e and e' leaked to the adversary. Formally,

$$\begin{aligned}\mathsf{exposed}(\mathcal{H}, e) \coloneqq\ & \exists e' : \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Mem}((\mathsf{key},e'),\mathtt{0})} \\ & \wedge \forall e'' \in [e'+1, e] \cup [e+1, e'] : \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Ch}((\mathsf{tok},e''),\mathtt{0}\to\mathtt{H})}\end{aligned}$$

The next predicate $\mathsf{ctxLeaked}(i, e)$ determines if the ciphertext containing the i-th value leaked during epoch e. Recall that the ciphertext may leak in three ways: 1) from one of the host's memories, 2) from the owner-to-host channel used to send the first encryption, and 3) from one of the host-to-owner channels used to retrieve the value.

$$\begin{aligned}\mathsf{ctxLeaked}(i, e) \coloneqq\ & \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Mem}((i,e),\mathtt{H})} \vee e \in \mathsf{epoch}(\mathcal{H}, \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Ch}((\mathsf{ctx},i),\mathtt{0}\to\mathtt{H})}) \\ & \vee \exists j : e \in \mathsf{epoch}(\mathcal{H}, \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Ch}((\mathsf{ctx},i,j),\mathtt{H}\to\mathtt{0})})\end{aligned}$$

Finally, the can-leak predicate $\mathfrak{L}^i(\mathcal{H})$ is true if and only if the ciphertext containing the i-th value leaked during an exposed epoch e.

$$\mathfrak{L}^i(\mathcal{H}) \coloneqq \exists e : \mathsf{exposed}(\mathcal{H}, e) \wedge \mathsf{ctxLeaked}(i, e) \tag{1}$$

Commitment problem. As currently defined, we cannot construct a simulator that makes the real and ideal worlds indistinguishable because of the so-called commitment problem: if the distinguisher leaks a ciphertext first and then the corresponding decryption key, then the simulator must first output a ciphertext (e.g. encryption of 0's) and afterwards output a key that explains it to the message it receives from $\mathsf{Storage}$. Without very long keys or strong assumptions, achieving this is often impossible [Nie02]. Therefore, we instead weaken the statement and tweak the can-leak predicate for the memories containing keys (and channels containing tokens, since keys can be recovered from token leaks) to disable such scenarios.[7]

Essentially, we want to enforce that if some ciphertext leaked in a non-exposed epoch e, then any action that flips e to exposed is disallowed. Flipping is formalized as

$$\mathsf{flip}(\mathcal{H}, \mathcal{E}_{id}, e) \coloneqq \neg\mathsf{exposed}(\mathcal{H}, e) \wedge\ \mathsf{exposed}(\mathcal{H} + \{\mathcal{E}_{id}\}, e)$$

This leads to the following can-leak predicate of the real-world memory (the predicate for channels $\mathsf{Ch}^{(\mathsf{tok},e),\mathtt{0}\to\mathtt{H}}$ is analogous).

$$\mathfrak{L}_{\mathsf{Mem}((\mathsf{key},e),\mathtt{0})}(\mathcal{H}) \leq \forall e' : \left(\nexists i : \mathsf{ctxLeaked}(i, e') \vee \neg\mathsf{flip}(\mathcal{H}, \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Mem}((\mathsf{key},e),\mathtt{0})}, e')\right) \tag{2}$$

After taking care of all these technical details, we are now ready to state our first main theorem, proved in Appendix B.2:

[7] Readers more familiar with game-based notions can think of the situation when adversary's corruption allowing to trivially decrypt the challenge. In a game, it simply loses (cannot corrupt). We reflect it closely by saying that a memory cannot leak. However, this hurts composition — we no longer make a statement about any environment, which corrupts arbitrarily, but one where the memory is never corrupted after the ciphertext leaks.

Theorem 1. *There exists an efficient simulator* sim *such that, assuming* ue *is* IND-ENC-CPA *secure,*

$$\texttt{ue-owner}^{\mathsf{O}}\texttt{ue-host}^{\mathsf{H}}\mathsf{R}^{\mathrm{UE}} \mathrel{\hat{\approx}} \mathrm{sim}^{\mathsf{E}}\mathsf{Storage}_{n_s,\vec{\mathfrak{L}},\vec{\mathfrak{I}},\mathcal{M},\texttt{ActSeq}},$$

where the real-world resources R^{UE} *are defined in Fig. 7 and have the additional restrictions from equation* (2)*, and the can-leak predicates* $\vec{\mathfrak{L}}$ *of* Storage *are defined in equation* (1)*.*

5 Resistance Against Randomness Corruption

We extend our construction from the previous section to offer better protection against bad randomness sources. The additional attack vector is modeled by requiring that all protocols are deterministic and assuming explicit injectable randomness resources in the real world. The achieved security is reflected by the can-leak predicates of the constructed storage resource Storage, which now depend on randomness-injected events.

5.1 Protocol

We start with the construction from Section 4 and assume that the update algorithm ue.upd of the underlying updatable encryption scheme is deterministic.[8] At a high level, we mitigate the effect of bad randomness by, first, making the store operation (which encrypts the message) deterministic. This is possible, because the protocol can access the cell index i, which is unique for each stored message. This means that we can derive encryption randomness as the PRF output on input i, where the secret seed is kept as part of the owner's current epoch key. Second, we derive the randomness for update generation by mixing the fresh random coins with the old epoch key. This way, the new epoch key is compromised only if both the old epoch key and the randomness are known to the adversary. In these situations the new key is inherently exposed, because the adversary can compute it herself.

More precisely, the protocol requires an updatable encryption scheme ue, a PRF $\mathcal{F}$ and a PRG $\mathcal{G}$. It proceeds as in Section 4, except the following. The epoch keys are now of the form (k, s, t), where k is the current key for ue, s is a seed for $\mathcal{F}$, and t is a random value. The protocol proceeds identically as in Section 4, except when storing a message in the i-th cell, the it calls ue.enc with randomness computed as $\mathcal{F}(s, i)$. Moreover, when generating an update, the owner fetches the current key is (k, s, t), chooses a random r and computes $(r', s', t') \leftarrow \mathcal{G}(t \oplus r)$ and $(k', \Delta) \leftarrow \mathsf{ue.nxt}(k; r')$. The new key is (k', s', t') and the update token is Δ.

[8] This is not a significant restriction. In fact, all IND-CCA secure schemes come with deterministic updates, because IND-CCA cannot be meaningfully defined otherwise.

This means that the protocol requires the same real-world memory and channel resources as the construction from Section 4, as well as n_u randomness resources for token generation and one randomness resource for generating the first epoch's key:

$$\mathsf{R}^{\texttt{UE-RAND}} \coloneqq [\mathsf{R}^{\texttt{UE}}, \{\mathsf{Rand}^e\}_{e\in[n_u]}, \mathsf{Rand}^{\mathsf{kg}}] \tag{3}$$

The protocol is implemented by the `ue-owner-rand` and `ue-host-rand` converters, which are plugged into the `O` and `H` interfaces of $\mathsf{R}^{\texttt{UE-RAND}}$, respectively. The converters are described in detail in Appendix C, albeit they are very similar to the converters of the simple model.

5.2 Construction Statement

The protocol constructs the ideal storage resource $\mathsf{Storage}$, which differs from the one constructed in Section 4 only in the can-leak predicates. In particular, we now use a different predicate to determine if an epoch is exposed: $\mathsf{exposedRand}$ is true if (as before, the key can be computed using leaked information or) the previous epoch is exposed and the randomness used in the "next" operation was injected. Moreover, injecting key-generation randomness exposes the first epoch.

$$\begin{aligned}\mathsf{exposedRand}(\mathcal{H}, e) \coloneqq \mathsf{exposed}(\mathcal{H}, e) \vee (e = 0 \ \wedge \ \mathcal{E}^{\mathsf{injected}}_{\mathsf{Rand(kg)}}) \\ \vee \ (\mathsf{exposedRand}(\mathcal{H}, e-1) \wedge \mathcal{E}^{\mathsf{injected}}_{\mathsf{Rand}(e)})\end{aligned}$$

Then, as before, the can-leak predicate of the i-th cell is true if an only if the ciphertext storing the i-th value leaked during an exposed epoch e:

$$\mathfrak{L}^i(\mathcal{H}) \coloneqq \exists e : \mathsf{exposedRand}(\mathcal{H}, e) \wedge \mathsf{ctxLeaked}(i, e) \tag{4}$$

Commitment problem. The can-leak predicate of the real-world memory needs the same restrictions as those introduced in the previous section, defined in *equation* (2). Further, note that, even though injecting randomness may result in epoch exposure, it will never be the case that this will trigger the commitment problem. This is due to the fact that a randomness injection can only expose a new epoch (the next one), for which ciphertexts evidently haven't leaked yet. So, there is no need to place additional restrictions on the can-inject predicate of the randomness resources.

Statement. We presented our protocol as a generic compiler but, unfortunately, no security property defined in the literature implies security of our protocol.

To be concrete, we construct a (contrived) scheme ue, which is IND-ENC-CPA secure, but makes our protocol completely insecure. The scheme ue modifies an IND-ENC-CPA secure scheme ue' as follows: $\mathsf{ue.nxt}$ called with randomness $\mathcal{G}(0)$ outputs a bad token $\Delta = (k, \mathtt{bad})$, where k is the old epoch key, and else executes $\mathsf{ue}'.\mathsf{nxt}$. To apply such token, the host decrypts the message and

stores it in plain as $(m, \texttt{plain})$. On input such "ciphertext", $\texttt{ue.upd}$ does nothing and $\texttt{ue.dec}$ outputs m. Clearly $\texttt{ue}'$ is IND-ENC secure, because the probability of honestly sampling $\mathcal{G}(0)$ is negligible. On the other hand, an adversary can attack our scheme by injecting $r = t$ (assuming she leaked the current key), which triggers the bad mode.

Therefore, we prove security of our protocol instantiated with the SHINE0 scheme [BDGJ20]. SHINE0 requires a group G where DDH is hard. In Appendix D we prove the following theorem (the activation sequences are the same as in Section 4).

Theorem 2. *Let G be a group and let* `ue-owner-rand` *and* `ue-host-rand` *be instantiated with SHINE0 and G. There exists an efficient simulator* sim *such that, assuming DDH in G hard, $\mathcal{F}$ is a secure PRF and $\mathcal{G}$ is a secure PRG,*

$$\texttt{ue-owner-rand}^{\mathtt{O}}\texttt{ue-host-rand}^{\mathtt{H}}\mathsf{R}^{\mathrm{UE}} \approx \mathrm{sim}^{\mathtt{E}}\mathsf{Storage}_{n_s,\vec{\mathfrak{L}},\vec{\mathfrak{I}},\mathcal{M},\texttt{ActSeq}},$$

where the real-world resources R^{UE} are defined in Fig. 7 and have the additional restrictions from equation (2)*, and the can-leak predicates of* Storage *are defined in equation* (4)*. Indistinguishability holds in the ideal-cipher model.*

6 Active Attackers

In this section, we expand our threat model from Section 4 and consider adversaries that are able to tamper with the host's memory (but not corrupt randomness). This is reflected by arbitrary can-inject predicates of the host's memories in the real world. We extend the protocol from Section 4 to provide confidentiality also in this setting — the constructed Storage has can-leak predicates the same as in Section 4, while its can-inject predicates are as good as those for host's memories in the real world.

In fact, the protocol needs only slight modifications. First, the UE scheme is replaced by one that is IND-ENC-RCCA secure (with arbitrary re-encryptions), e.g. the one from [KLR19]. Second, as described so far, the protocol is vulnerable to the following attack: copying ciphertexts from one memory to another would correspond in the ideal world to copying messages from one cell to another. Storage does not allow this (rightfully so, since the adversary essentially managed to modify a confidential storage, considered as a whole, into another related one). To prevent this attack, we modify the protocol as follows. To store message m in cell i, the owner encrypts (m, i). Then, if a value retrieved from cell i is not of the form (m', i), he outputs $\perp$, else m'.

The assumed resources are the same as in Section 4, except the can-leak predicates of $\mathsf{Mem}^{(i,e),\mathtt{H}}$ are arbitrary. The constructed Storage is also the same, except the can-inject predicate $\mathfrak{L}^i$ of the i-th cell is true if can-inject of $\mathsf{Mem}^{(i,e),\mathtt{H}}$ for the current epoch e is true. Formally, we let $\mathsf{currentEpoch}(\mathcal{H})$ be max e such that $\mathcal{E}^{\texttt{startUpdated:}e} \in \mathcal{H}$ and

$$\mathfrak{I}^i(\mathcal{H}) \coloneqq \mathfrak{I}_{\mathsf{Mem}((i,\mathsf{currentEpoch}(\mathcal{H})),\mathtt{H})}(\tilde{\mathcal{H}}) \tag{5}$$

The main result of this section is then the following theorem, proved in Appendix E:

Theorem 3. *There exists an efficient simulator* sim *such that, assuming* ue *is* IND-RCCA *secure and the message space* $\mathcal{M}$ *is large,*

$$\texttt{ue-owner-mall}^{\mathsf{O}}\texttt{ue-host-mall}^{\mathsf{H}}\mathsf{R}^{\mathrm{UE}} \mathrel{\hat{\approx}} \mathrm{sim}^{\mathsf{E}}\mathsf{Storage}_{n_s,\vec{\mathfrak{L}},\vec{\mathfrak{I}},\mathcal{M},\texttt{ActSeq}},$$

where the real-world resources R^{UE} *are defined in Fig. 7 and have the additional restrictions from equation* (2)*, and the can-leak and can-inject predicates of* Storage *are defined in equations* (1) *and* (5)*, respectively.*

7 Conclusions and Open Problems

We presented a framework for defining secure UE, which generalizes many previous definitions, as well as offers flexibility needed to discover new security-efficiency trade-offs. We took advantage of this new flexibility and constructed protocols with better security in the presence of randomness corruptions and active attackers, respectively.

Importantly, we discover that, first, various choices of the syntax have a negative effect on resistance against randomness corruption. Second, many security properties (at least ciphertext-independent) UE meant for active attackers only consider semi-active attacks, where the game allows the adversary to inject into ue.dec but not ue.upd. In our model, this means that the adversary can inject only into channels from the host to the owner and not in the opposite direction (thie would be equivalent to injecting into memory). This seems like a very weak and contrived setting.

We believe that our work opens new areas for research, some of which we list below.

Allowing arbitrary re-encryptions. Allowing active attackers to inject to ue.upd seems very important. It would be ideal to see more schemes (perhaps neither ciphertext-dependent nor independent) that provide authenticity and/or confidentiality in this setting.

Integrity protection. Our model can be very easily adjusted to model integrity protection — the only difference is that the adversary can always delete a message stored in Storage. We leave analyzing existing notions, as well as designing new, more efficient and secure protocols as future work.

Hiding meta-data. Often one would like to hide not only the message content, but also various meta-data, for instance, the message's "age", measured in the number of updates. We believe our model can be extended to formalize such guarantees.

Composition. We phrase our results in the constructive cryptography framework, which can provide composition of the strongest type: a protocol is secure in arbitrary environments, with arbitrary other protocols executed in parallel. This guarantee is very strong and we do not achieve it in this work (because of the so-called commitment problem, we give guarantees only in "most" environments). We leave constructing a protocol with the stronger guarantee (under sufficiently stronger assumptions) as future work.

References

[BDGJ20] Colin Boyd, Gareth T. Davies, Kristian Gjøsteen, and Yao Jiang. Fast and secure updatable encryption. pages 464–493, 2020.

[BEKS20] Dan Boneh, Saba Eskandarian, Sam Kim, and Maurice Shih. Improving speed and security in updatable encryption schemes. pages 559–589, 2020.

[BLMR13] Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. pages 410–428, 2013.

[Can01] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. pages 136–145, 2001.

[CLT20] Long Chen, Yanan Li, and Qiang Tang. CCA updatable encryption against malicious re-encryption attacks. pages 590–620, 2020.

[EPRS17] Adam Everspaugh, Kenneth G. Paterson, Thomas Ristenpart, and Samuel Scott. Key rotation for authenticated encryption. pages 98–129, 2017.

[Jia20] Yao Jiang. The direction of updatable encryption does not matter much. pages 529–558, 2020.

[JMM19] Daniel Jost, Ueli Maurer, and Marta Mularczyk. A unified and composable take on ratcheting. pages 180–210, 2019.

[KLR19] Michael Klooß, Anja Lehmann, and Andy Rupp. (R)CCA secure updatable encryption with integrity protection. pages 68–99, 2019.

[LT18] Anja Lehmann and Björn Tackmann. Updatable encryption with post-compromise security. pages 685–716, 2018.

[Mau02] Ueli M. Maurer. Indistinguishability of random systems. pages 110–132, 2002.

[Mau11] U Maurer. Constructive Cryptography–A New Paradigm for Security Definitions and Proofs. In *Theory of Security and Applications – TOSCA 2011*, pages 33–56. Springer Berlin Heidelberg, 2011.

[MPR07] Ueli M. Maurer, Krzysztof Pietrzak, and Renato Renner. Indistinguishability amplification. pages 130–149, 2007.

[MR11] Ueli Maurer and Renato Renner. Abstract cryptography. pages 1–21, 2011.

[Nie02] Jesper Buus Nielsen. Separating random oracle proofs from complexity theoretic proofs: The non-committing encryption case. pages 111–126, 2002.

[Rog04] Phillip Rogaway. Nonce-based symmetric encryption. pages 348–359, 2004.

[SF07] Dan Shumow and Niels Ferguson. On the possibility of a back door in the nist sp800-90 dualec prng. CRYPTO Rump Session, 2007.

[SS06] Berry Schoenmakers and Andrey Sidorenko. Cryptanalysis of the dual elliptic curve pseudorandom generator. Cryptology ePrint Archive, Report 2006/190, 2006. `http://eprint.iacr.org/2006/190`.

Supplementary Material

A Additional Preliminaries

A.1 Syntax

The appendix contains the formal definition of all converters and simulators. Just like for resources, we describe the converters and simulators using pseudo-code, and following the same conventions delineated in Section 2.3. We also assume that all arrays are 0-indexed, and we use the notation $A[i]$ to retrieve the i-th element of array A. So, our pseudocode closely resembles Python syntax.

A.2 Details of UE Security Notions

Formally, the `IND-ENC-CPA` and `IND-ENC-RCCA` games are defined in Figs. 8 and 9, respectively.

In `IND-ENC-RCCA`, the decryption oracle rejects any inputs that decrypt to either challenge message, and returns a special value `test` instead. Conversely, the Upd oracle does accept ciphertexts that decrypt to either challenge message, but it will add the current epoch to the list of epochs where the adversary knows the challenge ciphertext.

To determine if the adversary trivially wins, the challenger stores lists $\mathcal{K}$ and $\mathcal{T}$ of epochs for which, respectively, keys and tokens were corrupted by the adversary. Furthermore, recall that keys and tokens can be derived/inferred from corrupted material. So, the challenger also stores lists $\mathcal{K}^*$ and $\mathcal{T}^*$, extensions of $\mathcal{K}$ and $\mathcal{T}$, respectively, which also contain the inferred keys and tokens. More specifically, a key can be derived using the key from the prior/future epoch and the token between the two (i.e., it can be "upgraded" or "downgraded"). Conversely, tokens can be derived using both keys between the epochs when it was computed. Lastly, the challenger also maintains a list C^* of epochs in which the adversary learned a version of the challenge ciphertext (which is an extension of the list C of epochs in which the adversary called the UpdC oracle to honestly update the challenge ciphertext). That is, the adversary can update (bidirectionally) the challenge ciphertext locally using leaked (or derived) tokens. That is, if the ciphertext is encrypted with k_e, she can use Δ_{e+1} to get a ciphertext encrypted with k_{e+1} or she can use Δ_e to get a ciphertext encrypted with k_{e-1}.

Formally, the game is defined in Fig. 8. We say that the adversary trivially wins if $K^* \cap C^* \neq \varnothing$.

B Details of Section 4

B.1 Protocol Description

Recall that owner and host communicate via a series of single-use, unidirectional, authenticated channels. The first set of these are used to send new ciphertexts

Game IND-ENC-CPA

$\mathbf{Exp}^{\mathsf{IND\text{-}ENC\text{-}CPA}}_{\mathsf{ue},\mathcal{A}}()$

$b \leftarrow\$ \{0,1\}$
$k_0 \leftarrow\$ \mathsf{ue.kg}()$
$\Delta_0 \leftarrow \bot$
$e, n_c \leftarrow 0$
$\mathsf{phase} \leftarrow 0$
$\mathcal{L}, \tilde{\mathcal{L}}, \mathcal{C}, \mathcal{K}, \mathcal{T} \leftarrow \varnothing$
$(m_0, m_1, st) \leftarrow \mathcal{A}_1^{\mathbf{Enc,Nxt,Upd,Cor}}(\lambda)$
$\tilde{c} \leftarrow\$ \mathsf{ue.enc}(k_e, m_b)$
$b' \leftarrow \mathcal{A}_2^{\mathbf{Enc,Nxt,Upd,Cor,UpdC}}(st, \tilde{c})$
$b = b' \wedge \neg \texttt{trivial-win}(\mathcal{L}, \tilde{\mathcal{L}}\mathcal{K}, \mathcal{T}, \mathcal{C})$

Oracle Enc(m)

$c \leftarrow\$ \mathsf{ue.enc}(k_e, m)$
n_c**++**
$\mathcal{L}$ **+←** (n_c, c, e)
return c

Oracle Nxt

e**++**
$(k_e, \Delta_e) \leftarrow\$ \mathsf{ue.nxt}(k_{e-1})$
if $\mathsf{phase} = 1$ **then**
  $\tilde{c}_e \leftarrow \mathsf{ue.upd}(\Delta_e, \tilde{c}_{e-1})$

Oracle Upd(c_{e-1})

if $(j, c_{e-1}, e-1) \notin \mathcal{L}$ **then return** $\bot$
$c_e \leftarrow \mathsf{ue.upd}(\Delta_e, c_{e-1})$
$\mathcal{L}$ **+←** (j, c_e, e)
return c_e

Oracle Cor(inp, e'**)**

if $e' > e$ **then return** $\bot$
if $\mathsf{inp} = \mathsf{key}$ **then**
  $\mathcal{K}$ **+←** e'
  return $k_{e'}$
else if $\mathsf{inp} = \mathsf{token}$ **then**
  $\mathcal{T}$ **+←** e'
  return $\Delta_{e'}$

Oracle UpdC()

$\mathcal{C}$ **+←** e
$\tilde{\mathcal{L}}$ **+←** $(\tilde{c}_e, e)$
return $\tilde{c}_e$

Fig. 8: Description of the IND-ENC security game.

from the owner to the host during a `store` action; the second set of these are used to send a token from the owner to the host during an epoch change; the third set of these are used to send a "command" from the owner to the host (e.g., attempt to retrieve or delete a ciphertext); the fourth set of these are used to (re)send a stored ciphertext back to the owner during a `retrieve` action; the fifth (and last) set of channels are used to send ACK signals from the owner to the host, after an operation is completed. So, the high-level workflow of an action consists of the owner sending some message to the host via one of the aforementioned channels (and, thus, starting the action), followed by the host receiving this value and performing the appropriate internal computations, (potentially) followed by the host sending a ciphertext back to the owner (if and only if the action taking place is a retrieval), followed by the host sending an ACK signal, followed by the owner receiving this signal and retrieving the ciphertext sent by the host (if any). In particular, note that all of the host's activity happens inside/during its `activate` input, and the owner receives the ACK signal and finishes the action during its `activate` input. Looking at the behavior of the converters clearly illustrates the need for the `activate` inputs: each action is an interactive protocol. Thus, we need inputs on either side that encapsulate the rounds of interaction (and block further inputs until the other side finishes its round). The formal description of the converters is in Figs. 10 and 11.

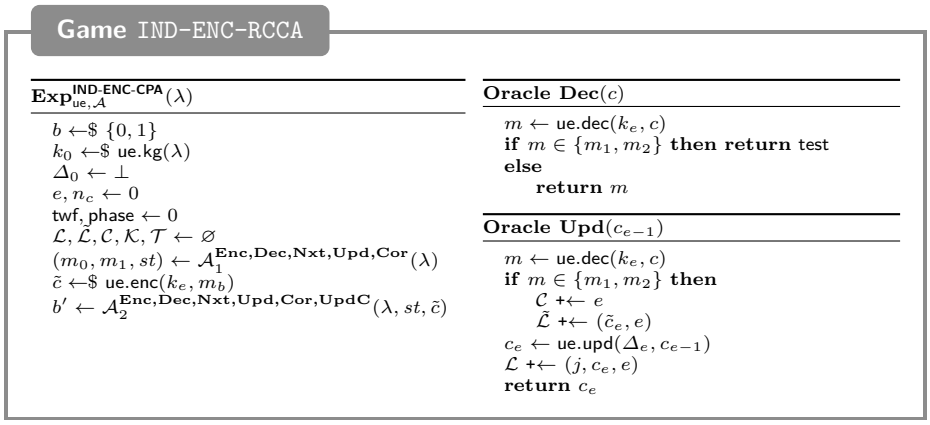

Fig. 9: Description of the RCCA security game.

B.2 Proof of Theorem 1

Theorem 1. *There exists an efficient simulator* sim *such that, assuming* ue *is* IND-ENC-CPA *secure,*

$$\mathtt{ue\text{-}owner}^{\mathtt{O}}\mathtt{ue\text{-}host}^{\mathtt{H}}\mathsf{R}^{\mathrm{UE}} \approx \mathrm{sim}^{\mathtt{E}}\mathsf{Storage}_{n_s,\vec{\mathfrak{L}},\vec{\mathfrak{I}},\mathcal{M},\mathtt{ActSeq}},$$

where the real-world resources R^{UE} *are defined in Fig. 7 and have the additional restrictions from equation* (2)*, and the can-leak predicates* $\vec{\mathfrak{L}}$ *of* Storage *are defined in equation* (1)*.*

Simulator. The simulator $\mathsf{sim}_{\mathtt{UE}}$ is described in detail in Fig. 12. At a high level, $\mathsf{sim}_{\mathtt{UE}}$ simply executes the protocol, except (fresh) encryptions of values not revealed by Storage are generated by encrypting a vector of 0's of the length revealed by the output of leakLength. To do this, the simulator first samples a secret key with UE.setup. Then, it maintains an array of keys tokens, and ciphertexts that gets extended, as needed, with UE.next. Since all these keys and tokens are sampled honestly, they have the same distribution as the ones in the real world (i.e., an adversary seeing a subset of the keys from either world can't know which world she is seeing). For simulating the ciphertexts, we use the standard strategy of encrypting the message, if possible, or a string of zeroes of the same length. That is, the simulator will try to leak the message and use the appropriate key from the array (or generate it) to encrypt it. If the can-leak predicate of FMem doesn't allow it to leak the message it will leak the length of the message (which is always possible), and encrypt a string of zeroes of this length instead. Further, the simulator maintains an array C of ciphertexts, where the i-th entry is the ciphertext on memory cell i. Whenever a new ciphertext gets leaked for the first, the simulator adds the generated ciphertext to C, and any future leaks simply retrieve the value from it. In addition, on every epoch

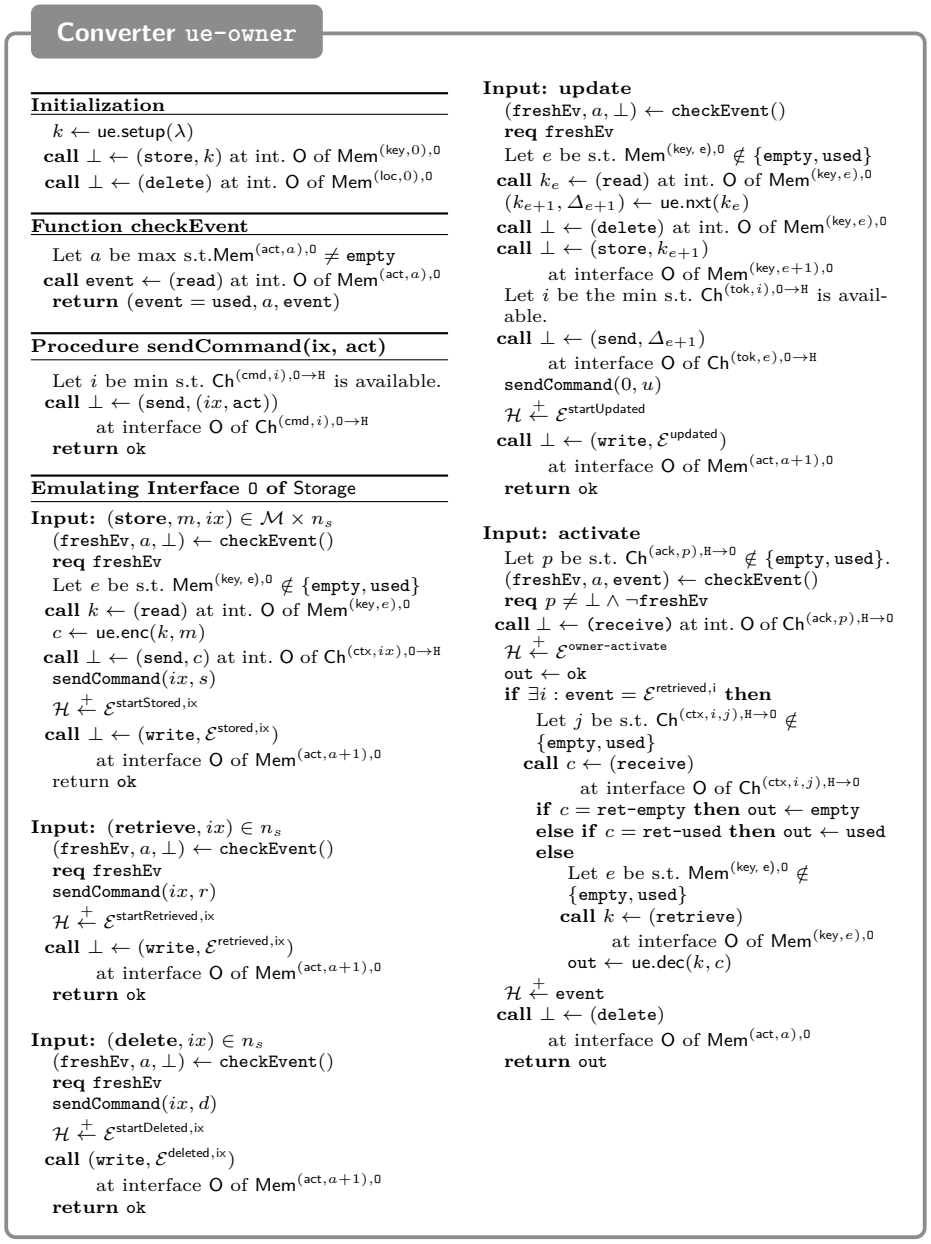

Fig. 10: Description of the owners' converter for the simple model.

Converter `ue-host`

Emulating Interface `H` **of** $\mathsf{Storage}_{0,\mathtt{H}}$

Input: activate
Let p be s.t. $\mathsf{Ch}^{(\mathsf{cmd},p),0\to\mathtt{H}} \notin \{\mathtt{empty},\mathtt{used}\}$
req $p \neq \bot$
call $(i,\mathtt{cmd}) \leftarrow (\mathtt{receive})$ at int. H of $\mathsf{Ch}^{(\mathsf{cmd},p),0\to\mathtt{H}}$
Let e be max s.t. $\mathsf{Ch}^{(\mathsf{tok},e),0\to\mathtt{H}} = \mathtt{used}$ or -1 if no such e.
if $\mathtt{cmd} = s$ **then**
  call $c \leftarrow (\mathtt{receive})$ at int. H of $\mathsf{Ch}^{(\mathsf{ctx},i),0\to\mathtt{H}}$
  if $c \neq \mathtt{ok}$ **then**
    call $\bot \leftarrow (\mathtt{write}, c)$ at int. H of $\mathsf{Mem}^{(i,e+1),\mathtt{H}}$
else if $\mathtt{cmd} = r$ **then**
  call $c \leftarrow (\mathtt{retrieve})$ at int. H of $\mathsf{Mem}^{(i,e+1),\mathtt{H}}$
  if $c \in \mathtt{empty},\mathtt{used}$ **then** $c = \mathsf{ret}\text{-}c$
  Let j be min s.t. $\mathsf{Ch}^{(\mathsf{ctx},i,j),\mathtt{H}\to 0}$ is available.
  call $\bot \leftarrow (\mathtt{send}, c)$ at int. H of $\mathsf{Ch}^{(\mathsf{ctx},i,j),\mathtt{H}\to 0}$
else if $\mathtt{cmd} = d$ **then**
  call $\bot \leftarrow (\mathtt{erase})$ at int. H of $\mathsf{Mem}^{(i,e),\mathtt{H}}$
else
  call $\Delta_e \leftarrow (\mathtt{receive})$ at int. H of $\mathsf{Ch}^{(\mathsf{tok},e+1),0\to\mathtt{H}}$
  for $\mathtt{ix} \in [n_s]$ **do**
    call $c \leftarrow (\mathtt{read})$ at int. H of $\mathsf{Mem}^{(\mathtt{ix},e+1)\mathtt{H}}$
    if $c \notin \{\mathtt{empty},\mathtt{used}\}$ **then**
      call $\bot \leftarrow (\mathtt{write}, \mathsf{ue.upd}(c, \Delta_e))$ at int. H of $\mathsf{Mem}^{(\mathtt{ix},e+2),\mathtt{H}}$
      call $\bot \leftarrow (\mathtt{erase})$ at int. H of $\mathsf{Mem}^{(\mathtt{ix},e+1),\mathtt{H}}$
$\mathcal{H} \overset{+}{\leftarrow} \mathcal{E}^{\mathtt{host\text{-}activate}}$
Let p be min s.t. $\mathsf{Ch}^{(\mathsf{ack},p),\mathtt{H}\to 0}$ is available.
call $\bot \leftarrow (\mathtt{send}, 1)$ at int. H of $\mathsf{Ch}^{(\mathsf{ack},p),\mathtt{H}\to 0}$
return `ok`

Fig. 11: Description of the host's converter for the simple model.

change, all (non-$\perp$) ciphertexts in C get updated, enforcing the fact that C always contains ciphertexts that are encrypted under the latest key.

Consistency of events. We next show that the event histories in the real and in the ideal worlds are consistent. The ideal-world events are mapped to the real-world events by the mapping τ that sends $\mathcal{E}^{\texttt{event}}_{\mathsf{Storage}}$ to $\mathcal{E}^{\texttt{event}}_{\texttt{ue-owner}}$, for `event` in {(`start-store, i`), (`start-retrieve, i, j`), (`start-delete, i`), `start-update`, (`store, i`), (`retrieve, i`), (`delete, i`), `update`, `owner-activate`}. In addition, τ maps $\mathcal{E}^{\texttt{host-activate}}_{\mathsf{Storage}}$ to $\mathcal{E}^{\texttt{host-activate}}_{\texttt{ue-host}}$. Other events get mapped by τ to themselves.

To show that τ yields consistent event histories, we first note that the adversary's leaked events are clearly triggered at the same time. Moreover, in both worlds the history can be partitioned into consecutive segments, where in each segment all non-adversarial events relevant for τ correspond to one action (this easily follows by inspection). Therefore, we now analyze each segment separately:

– *Action store.* In both worlds, this action starts with a `(store, m, i)` input to the `O` interface, which returns `ok` and triggers equivalent (modulo τ) `start-store` events. The `O` interface is now "blocked", returning $\perp$ for any input and triggering no events. The ideal world enforces this by checking the activation dictionary, and the real world enforces this using an `event` local variable, stored in a memory, which blocks all inputs until it is cleared at the end of the action. The `activate` input is the only exception: instead of using the `event` local variable, it gets unblocked when it receives an ACK signal from the host. The `H` interface was blocked until now: the ideal world, again, uses the interleaving dictionary, and the real world returns $\perp$ until one of the command channels is available (which happens if and only if an action was initiated by the owner; in this case, a ciphertext channel is available). However, the `H` interface is now active, accepting (only) `activate` inputs. In both worlds, `ok` is returned and a `host-activate` is triggered. This inactivates the `H` interface (since, as before, no new channels are available), and enables the `O` interface only for the `activate` input (all other interfaces return $\perp$, since they are still blocked from the first input). As mentioned earlier, in the real world, at the end of the previous input (the host's `activate`) an ACK signal is sent to the owner, which unblocks the `activate` input. In the ideal world, the (un)blocking is still enforced with the interleaving dictionary. Lastly, the owner finally inputs `activate`, which triggers `owner-activate` and `(store, i)` events (in that order), and returns `ok`. This blocks the `activate` input for now (since no key in the interleving dictionary has this value, and there are no empty ACK channels), but unblocks all other inputs in the `O` interface (since there is a key for each of them in the interleaving dictionary, and the `event` variable got wiped back to $\perp$). Importantly, note that attempting to store a message in an unavailable cell fails silently in both worlds. That is, the owner and host still go through the sequence of activations, but the store input to the memory cell (in both worlds) simply ignores the input, but doesn't fail. After this lengthy analysis, we can see that both

Simulator $\mathsf{sim}_{\mathtt{UE}}$

Initialization

$k_0 \leftarrow \mathsf{ue.kg}()$
$K \leftarrow [k_0]$
$T \leftarrow []$
$C \leftarrow [\bot \text{ for i in range}(n_s)]$
$L \leftarrow [\bot \text{ for i in range}(n_s)]$

Emulating Interface E **of** $\mathsf{Mem}^{(i,e),\mathtt{H}}$

Input: leak

req $\neg\mathsf{isMemEmpty}(i,e) \wedge \mathfrak{L}^{R^{\mathrm{UE}}}_{\mathsf{Mem}(i,\mathtt{H})}(\tilde{\mathcal{H}})$
$\mathtt{updateState}(i, \mathsf{Mem}((i,e),\mathtt{H}))$
return $C[i]$

Input: leakLength

if $\mathsf{isMemUnused}(i)$ **then return** empty
else if $\mathsf{isMemDeleted}(i,e)$ **then return** used
else
  $\mathtt{updateState}(i, \mathsf{length})$
  return $|L[i]|$

Emulating Interface E **of** $\mathsf{Mem}^{(\mathtt{key},e),\mathtt{0}}$

Input: leak

req $e = \mathsf{currentEpoch}() \wedge \mathfrak{L}^{R^{\mathrm{UE}}}_{\mathsf{Mem}(\mathsf{key},\mathtt{0})}(\tilde{\mathcal{H}})$
$\mathtt{updateState}()$
$\mathcal{H} \xleftarrow{+} \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Mem}((\mathsf{key},e),\mathtt{0})}$
return $|K[e]|$

Input: leakLength

$E \leftarrow \mathsf{currentEpoch}()$
if $e > E$ **then return** empty
else if $e < E$ **then return** used
else
  $\mathtt{updateState}()$
  return $|K[e]|$

Emulating Interface E **of** $\mathsf{Mem}^{(\mathtt{act},a),\mathtt{H}}$

Input: leak

$o \leftarrow \mathtt{findCommand}(p)$
req $o \neq \bot \wedge \mathfrak{L}^{R^{\mathrm{UE}}}_{\mathsf{Mem}((\mathsf{act}),\mathtt{H})}(\tilde{\mathcal{H}})$
return $\mathcal{E}^{\circ}_{\mathsf{Storage}}$

Input: leakLength

call $o \leftarrow (\mathtt{leak})$ at int. E of $\mathsf{Mem}^{(\mathtt{act},a),\mathtt{0}}$
return $|o|$

Emulating Interface E **of** $\mathsf{Ch}^{(\mathtt{ctx},i),\mathtt{0}\to\mathtt{H}}$

Input: leak

req $\neg\mathsf{isChEmpty}(\mathtt{ctx}, i) \wedge \mathfrak{L}^{R^{\mathrm{UE}}}_{\mathsf{Ch}((\mathsf{ctx},i),\mathtt{0}\to\mathtt{H})}(\tilde{\mathcal{H}})$
$e \leftarrow \mathsf{epoch}(\mathcal{E}^{\mathtt{startStored},i:0}_{\mathsf{Storage}})$
$\mathtt{updateState}(i, \mathsf{Ch}((\mathsf{ctx}, i), \mathtt{0} \to \mathtt{H}))$
return $C[i]$

Input: leakLength

if $\neg\mathcal{E}^{\mathtt{startStored},i:0}_{\mathsf{Storage}}$ **then return** empty
else if $\neg\mathsf{isMemUnused}(i)$ **then return** used
else
  $\mathtt{updateState}(i, \mathsf{length})$
  return $|L[i]|$

Emulating Interface E **of** $\mathsf{Ch}^{(\mathtt{cmd},p),\mathtt{0}\to\mathtt{H}}$

Input: leak

$(a, i) \leftarrow \mathtt{findCommand(p)}$
req $(a,i) \neq \bot \wedge \neg\mathcal{E}^{(\mathtt{a},\mathtt{i})}_{\mathsf{Storage}} \prec \mathcal{E}^{\mathtt{host\text{-}activate}}_{\mathsf{Storage}}$
  $\wedge\ \mathfrak{L}^{R^{\mathrm{UE}}}_{\mathsf{Ch}((\mathsf{cmd},p),\mathtt{0}\to\mathtt{H})}(\tilde{\mathcal{H}})$
return $(a[\mathtt{0}], i)$

Input: leakLength

call $o \leftarrow (\mathtt{leak})$ at int. E of $\mathsf{Ch}^{(\mathsf{cmd},p),\mathtt{0}\to\mathtt{H}}$
return $|o|$

Emulating Interface E **of** $\mathsf{Ch}^{(\mathtt{tok},e),\mathtt{0}\to\mathtt{H}}$

Input: leak

req $\neg\mathsf{isChEmpty}(\mathtt{tok}, e) \wedge \mathfrak{L}^{R^{\mathrm{UE}}}_{\mathsf{Ch}((\mathsf{tok},e),\mathtt{0}\to\mathtt{H})}(\tilde{\mathcal{H}})$
$\mathcal{H} \xleftarrow{+} \mathcal{E}^{\mathsf{leaked}}_{\mathsf{Ch}((\mathsf{tok},e),\mathcal{T},\mathtt{0}\to\mathtt{H})}$
$\mathtt{updateState}()$
return $T[e]$

Input: leakLength

if $\neg\ \mathcal{E}^{\mathtt{startUpdated:}e}_{\mathsf{Storage}}$ **then return** empty
else if $\mathcal{E}^{\mathtt{startUpdated:}e}_{\mathsf{Storage}} \prec \mathcal{E}^{\mathtt{host\text{-}activate}}_{\mathsf{Storage}}$ **then**
  return used
else
  $\mathtt{updateState}()$
  return $|T[e]|$

Emulating Interface E **of** $\mathsf{Ch}^{(\mathtt{ctx},i,j),\mathtt{H}\to\mathtt{0}}$

Input: leak

req$\neg\mathsf{isChEmpty}(\mathtt{ctx}, i, j) \wedge$
$\mathfrak{L}^{R^{\mathrm{UE}}}_{\mathsf{Ch}((\mathsf{ctx},i,j),\mathtt{0}\to\mathtt{H})}(\tilde{\mathcal{H}})$
$\mathtt{updateState}(i, \mathsf{Ch}((\mathsf{ctx}, i, j), \mathtt{0} \to \mathtt{H}))$
return $C[i]$

Input: leakLength

if $\neg\mathcal{E}^{\mathtt{startRetrieved},i:j}_{\mathsf{Storage}} \prec \mathcal{E}^{\mathtt{host\text{-}activate}}_{\mathsf{Storage}}$ **then**
  return empty
else if $\mathcal{E}^{\mathtt{retrieved},i:j}_{\mathsf{Storage}}$ **then return** used
else
  $\mathtt{updateState}(i, \mathsf{length})$
  return $|L[i]|$

Emulating Interface E **of** $\mathsf{Ch}^{(\mathtt{ack},p),\mathtt{H}\to\mathtt{0}}$

Input: leak

Let $\mathcal{E}'$ be the p-th s.t. $\mathcal{E}^{\mathtt{host\text{-}activate}}_{\mathsf{Storage}}$
req $\mathcal{E}' \neq \bot \wedge \neg\mathcal{E}' \prec \mathcal{E}^{\mathtt{owner\text{-}activate}}_{\mathsf{Storage}}$
  $\wedge\ \mathfrak{L}^{R^{\mathrm{UE}}}_{\mathsf{Ch}((\mathsf{ack},p),\mathtt{H}\to\mathtt{0})}(\tilde{\mathcal{H}})$
return 1

Input: leakLength

call $o \leftarrow (\mathtt{leak})$ at int. E of $\mathsf{Ch}^{(\mathsf{ack},p),\mathtt{0}\to\mathtt{H}}$
return $|o|$

Fig. 12: Formal definition of the simulator. Additional procedures are defined in Fig. 13. Recall that after any start event, the only possible (non-adversarial) event is an activation from the host. Further, seeing this event implies that the messages sent from the owner to the host are no longer on their channels.

Simulator $\mathsf{sim}_{\mathtt{UE}}$: Additional Procedures

Procedure currentEpoch()
  Let e be max s.t. $\mathcal{E}^{\mathtt{startUpdated}:e}_{\mathsf{Storage}}$
    or -1 if no such event occurred
  return $e+1$
Procedure epoch($\mathcal{E}^{\mathsf{id}}$)
  return max e s.t. $\mathcal{E}^{\mathtt{updated}:e}_{\mathsf{Storage}} \prec \mathcal{E}^{\mathsf{id}}$
    or 0 if such e exists
Procedure isMemUnused(i)
  return $\neg\mathcal{E}^{\mathtt{startStored},i:0}_{\mathsf{Storage}} \prec \mathcal{E}^{\mathtt{host\text{-}activate}}_{\mathsf{Storage}}$
Procedure isMemDeleted(i, e)
  return $e \neq$ currentEpoch()
    $\vee\ \mathcal{E}^{\mathtt{startDeleted},i:0}_{\mathsf{Storage}} \prec \mathcal{E}^{\mathtt{host\text{-}activate}}_{\mathsf{Storage}}$
Procedure isMemEmpty(i, e)
  return isMemUnused(i) $\vee$ isMemDeleted(i, e)
Procedure isChEmpty($\mathtt{tok}, e$)
  return $\neg\ \mathcal{E}^{\mathtt{startUpdated}:e}_{\mathsf{Storage}}$
    $\vee\ \mathcal{E}^{\mathtt{startUpdated}:e}_{\mathsf{Storage}} \prec \mathcal{E}^{\mathtt{host\text{-}activate}}_{\mathsf{Storage}}$
Procedure isChEmpty($\mathtt{ctx}, i$)
  return $\neg\mathcal{E}^{\mathtt{startStored},i:0}_{\mathsf{Storage}} \vee \neg$isMemUnused($i$)
Procedure isChEmpty($\mathtt{ctx}, i, j$)
  return $\neg\mathcal{E}^{\mathtt{startRetrieved},i:j}_{\mathsf{Storage}} \prec \mathcal{E}^{\mathtt{host\text{-}activate}}_{\mathsf{Storage}}$
    $\vee\ \mathcal{E}^{\mathtt{retrieved},i:j}_{\mathsf{Storage}}$
Procedure findCommand(p)
  $t \leftarrow 0$
  $\mathtt{l} \leftarrow$ {Stored, Retrieved, Deleted, Updated}
  if $p < \mathtt{len}(l)$ **then return** $\perp$
  for $\mathcal{E}' \in \mathcal{H}$ **do**
    if $\exists \mathtt{act} \in \mathtt{l}, i : \mathcal{E}' = \mathcal{E}^{\mathtt{startAct},\mathtt{ix}}$ **then**
      if t = p **then return** $(\mathtt{a}, i)$
      else t++
  return $\perp$

Procedure updateState($i = \perp, \mathtt{R} = \perp$)
  $e \leftarrow$ currentEpoch()
  $l \leftarrow \mathtt{len}(K)$
  for $j \in [l, e]$ **do**
    $(K[j], \Delta) \leftarrow$ ue.nxt($K[\mathtt{-1}]$)
    $T[j] \leftarrow \Delta$
    for i s.t. $C[i] \neq \perp$ **do**
      $C[i] \leftarrow$ ue.upd($\Delta, C[i]$)
    for l s.t. $L[l] \neq \perp$ **do**
      $L[l] \leftarrow$ ue.upd($\Delta, L[l]$)
  if $\mathtt{R} \notin$ {length, $\perp$} **then**
    if $C[i] = \perp$ **then**
      $k \leftarrow K[\mathtt{-1}]$
      call $m \leftarrow$ (leak, i) at int. E of Storage
      if $m = \perp$ **then**
        $\mathcal{H} \xleftarrow{+} \mathcal{E}^{\mathsf{leaked}}_{\mathtt{R}}$
        call $l \leftarrow$ (leakLength, i)
            at interface E of Storage
        $m = 0^l$
      $c \leftarrow$ ue.enc(k, m)
      $C[i] \leftarrow c$
      $L[i] \leftarrow c$
    else
      $\mathcal{H} \xleftarrow{+} \mathcal{E}^{\mathsf{leaked}}_{\mathtt{R}}$
  else if $\mathtt{R} =$ length $\wedge\ L[i] = \perp$ **then**
    call $l \leftarrow$ (leakLength, i)
        at interface E of Storage
    $c \leftarrow$ ue.enc($k, 0^l$)
    $L[i] \leftarrow c$
  return

Fig. 13: Additional procedures used by the simulator.

worlds are indistinguishable during the execution of this action: equivalent events are triggered and the same values are returned on all inputs at all times (including $\perp$, meaning that inputs get blocked and unblocked in the same manner).

- *delete.* In both worlds, this action starts with a `(delete, i)` input to the `O` interface. If no message has been stored yet, the cell is now inactive in both worlds, by virtue of the internal value being set to `used` a priori. Conversely, if i is contains a message, the action proceeds exactly as `store`, triggering analogous events. The only subtle difference is that the channel that gets activated (which, in turn, unblocks the host's `activate` input), is now a "command" channel instead of a ciphertext channel. Thus, by the same argument, both worlds are indistinguishable during the execution of this action.
- *retrieve* In both worlds, this action starts with a `(retrieve, i)` input to the `O` interface. The analysis and behavior is the same as the delete action, except for the last return value (after the owner's `activate`): instead of returning `ok`, the i-th stored message is returned. In the ideal world, the i-th message is retrieved directly (conceptually, this corresponds to forwarding the `retrieve` input to the i-th memory cell). In the real world, at the end of the host's `activate`, it sends the i-th ciphertext via a channel back to the owner. Then, in the owner's `activate` (the next input), this value is retrieved and decrypted, and thus the same message is retrieved. Note that, even though this ciphertext may not be the same one that was originally stored (since epochs may have changed) the correctness guarantees of the UE scheme ensure that this decryption returns the correct message. Further, note that in both worlds, trying to retrieve a value from an empty cell fails silently: either `empty` or `used` is returned. So, it follows that the behavior of both worlds is indistinguishable once again.
- *update* In both worlds, this action starts with an `update` input to the `O` interface. This action follows the same framework from the previous ones, so the indistinguishability argument is the same. One important thing to note, however, is that the internal computations of this action are particularly relevant: after the first input, the `O` converter rotates the key and sends the update token to the host, via a dedicated channel. Then, the `H` converter updates all ciphertexts using this new token. The fact that these computations take places is precisely what guarantees that the owner receives the same message after a retrieval, even if multiple epoch changes have taken place.

Indistinguishability. Finally, we show that if the updatable encryption scheme is IND-ENC secure, then the outputs at the `E` interface in the real and ideal world are indistinguishable. To this end, we first show that the outputs from each resource are of the same type, e.g. a ciphertext or $\perp$, ignoring what is encrypted.

We now proceed to go over every simulated resource, and how the outputs agree. with those in the real world. Note that, in all resources below, $\perp$ is returned in both worlds if the can-leak predicate of the (real world) resource is false (the

simulator explicitly checks this condition). As such, the analysis below will focus on the scenarios where the predicate is true.

– $\mathsf{Mem}^{(i,e),\mathcal{C},\mathtt{H}}$. We will describe this first entry in greater detail, since it serves as a template for all others.
If the message is not stored or if it was deleted, this operation returns $\perp$. More specifically, $\perp$ is returned if there has not been a `store` input for the i-th message directly followed by an activation from the host (with any arbitrary adversarial inputs in between, of course). A `store` input is not enough, since the ciphertext doesn't reach the memory cell until the host receives it (i.e., an activation). Similarly, $\perp$ is returned if there was a `delete` input for that cell, immediately followed by an activation from the host. As before, the `delete` input is not enough, since the ciphertext doesn't actually get deleted from the cell until the host receives the deletion command. So, it follows that an adversary could retrieve a ciphertext even after a delete action was started, and could fail to do so even after a store action started; timing is crucial. The protocol enforces these scenarios by construction (the aforementioned arguments correspond to the real world resources), and the simulator enforces them by explicitly checking the global event history for the events that correspond to these inputs.

  Conversely, if the message is present, the simulator leaks the message and generates a ciphertext for a message of the same length as the i-th stored one (using the strategy defined above) and, in the real world, the memory leaks the ciphertext itself. As such, in both cases, a distinguisher sees an encryption of a message of the same length, under a valid key of the UE world for that epoch (note how the simulator samples all keys up to the needed epoch). The RW is in the e-th epoch explicitly (as the converters rotate keys every update operation), and the IW is in it implicitly (every update operation, an event is triggered, so we can retrieve the epoch number via the global event history). As such, applying the same number of updates in both worlds results in the same epoch number being used.

– $\mathsf{Mem}^{(\mathsf{key},e),\mathcal{K},\mathtt{0}}$. Recall that the current secret key gets deleted after an epoch change. As such, only one of these memories contains a key at a given time (namely, the memory whose index corresponds to the current epoch). In both worlds, if this memory is not empty, a valid key (for the same epoch) is leaked. In the real world, the owner stored the secret key in it, so it can be leaked directly. In the ideal world, as mentioned earlier, the simulator maintains an array of honestly sampled keys. If it already has a key for the current epoch (computed during some other input), this can be leaked. If not, the simulator sequentially computes a new key for every epoch between the latest key's epoch and the present one, using the UE.next algorithm and the previous key, and stores all of these in the array, before releasing the last one. Conversely, if the memory does not correspond to a current epoch, $\perp$ is returned. In the real world, since this is an old epoch, the owner wiped

the memory when it transitioned to the next epoch. In the ideal world, the simulator compares the global epoch and the memory's index, and returns $\perp$ if these values don't match.

- $\mathsf{Ch}^{(\mathsf{tok},e),\mathcal{T},\mathtt{O}\to\mathtt{H}}$. Recall that channels get cleared after the message is received. Thus, unless the adversary leaks the contents of the channel between the time the message is sent and when it is received, $\perp$ will be returned. In this case, the adversary needs to leak the message after the owner starts an epoch change for that channel's ID (`update` input) and before the host receives it the update token (an activation by the host). To match this behavior, the simulator, as before, simply checks the global event history for the presence of an event indicating that an update for this channel's epoch started, but that it is not followed by an event indicating the host triggered an activation. If the timing is right, the adversary can leak the token from the channel. In the ideal world, as in the key memory, the simulator leaks the token from the array it has been maintaining, or computes it if needed. In both cases, the adversary sees a token if and only if it triggered this input at the exact same time.

- $\mathsf{Ch}^{(\mathsf{ctx},i),\mathcal{T},\mathtt{O}\to\mathtt{H}}$. The analysis is exactly the same as the token channel, but with ciphertexts being leaked instead. The simulator uses the same strategy as in the memories in order to generate the ciphertexts.

- $\mathsf{Ch}^{(\mathsf{cmd},p),\mathtt{O}\to\mathtt{H}}$. The adversary can leak the contents of this channel if her input is triggered after the owner starts an action, but before the host activates to process the operation. To enforce this, the simulator simply checks if the last (non-adversarial) event in the global event history is a start event for one of these three actions. If so, based on the ID of the event, it can extract the contents of the command sent and return this, which is exactly what is seen in the real world. If, conversely, the timing is off, the simulator returns $\perp$, just like it happens in the real world.
- $\mathsf{Ch}^{(\mathsf{ctx},i,j),\mathcal{T},\mathtt{H}\to\mathtt{O}}$. Since this channel goes in the opposite direction, the timing is a bit different. Namely, the leak must occur after the host's activation (which followed the owner's `retrieve` input) but before the owner's activation, which completes the action. If this is the case, a ciphertext is generated, using the standard strategy. If timing is off, $\perp$ is returned.
- $\mathsf{Ch}^{(\mathsf{ack},p),\mathtt{H}\to\mathtt{O}}$. The analysis is the same as the prior channel. The only difference is that this resource is used on every action, instead of only during a retrieval. As such, the simulator verifies that the host's activation when using this channel was not followed by the owner's activation (i.e., the channel has not been cleared, and the action has not been completed). If this is the case, the adversary leaks an ACK bit.

With this list, we can see that the `E` interface in both worlds yields indistinguishable behavior, both in terms of events and inputs/outputs. The last technical point to discuss is the IND-ENC security of the UE scheme. As mentioned earlier, it may be the case that, even though a message is present, the

simulator cannot forge an encryption of it (if can-leak predicate is false), and needs to encrypt a same-length string of zeroes instead. Hence, these two scenarios should be indistinguishable, assuming arbitrary corruptions of keys and tokens by the adversary. This is (almost) the precise definition of IND-ENC security, with a few small differences. First, the IND-ENC security game allows for only one challenge ciphertext, whereas the simulator may need to simulate more than one ciphertext. Secondly, we need to make sure that our fix for the commitment problem and can-leak predicates are compatible with the trivial wins of IND-ENC. That is, these disallow the trivial wins in the game. If an IND-ENC adversary uses the distinguisher as a subroutine, we need to make sure that this distinguisher will not be able to trigger trivial wins, as this would imply the adversary loses the game, and the reduction doesn't hold.

Every time the simulator needs to "fake" a ciphertext (encrypt a same-length string of zeroes) corresponds a pair of challenge messages in the IND-ENC security game. So, the UE scheme needs to be secure when the adversary is allowed to submit multiple challenge pairs, instead of just one. However, this reduction is simply the standard hybrid argument that proves that CPA security with one challenge implies CPA security with many challenge rounds.

For the second issue, recall that the trivial wins in the IND-ENC game are those where the adversary learns the secret key (either directly, by corrupting it, or indirectly, using corrupted tokens) in some epoch where she learned a version of one of the challenge ciphertexts. Note that it doesn't matter if the adversary learns the key first or the ciphertext first; as long as she eventually knows both, this is a trivial win. However, for our purposes, we only care about instances where she learns the ciphertext first and then the key. This is because the simulator only needs to "fake" ciphertexts in this scenario: when it commits to a ciphertext a priori, and then exposes the epoch key. If the key is corrupted first, the `can-leak` predicate of the memory cell is true, so the message can be leaked and there is no need to fake it, so this would not be a challenge ciphertext. For the other instances which we do care about, our fix for the commitment problem prevents the distinguisher from reaching this situation. Recall that our fix for the commitment problem blocks leakages of keys and tokens which "flip" an epoch, in which a ciphertext leaked, from uncompromised to compromised. Then, note that our *exposed* predicate encapsulates exactly the list $\mathcal{K}^*$ of epochs in which the the adversary learns the secret key in the IND-ENC security game, assuming bi-directional updates: $\mathcal{K}^*$ includes both direct corruptions of the key via an oracle, and indirect corruptions of it by updating a previously leaked key using corrupted tokens. Similarly, our *exposed* predicate checks if a key leaked for the current epoch (direct corruption), or if a key leaked from some other epoch that is reachable by a series of leaked tokens (indirect corruption). So, an epoch e is in $\mathcal{K}^*$ if and only if $exposed(\mathcal{H}, e)$ is true. Further, even though we only explicitly check for epochs in which a ciphertext leaked, whereas the IND-ENC game also considers epochs to which the adversary can (locally) update ciphertets, our commitment problem fix is still consistent: the later scenario is simply an extension of the former scenario, where the "chain of corruption"

gets extended by the tokens between the corruption of the ciphertext and the corruption of the epoch. This is still taken into account in our predicate. So, our fix for the commitment problem blocks the distiguisher from leaking keys or tokens that would result in a trivial win for the adversary, as desired.

With these two technical details resolved, it follows that the real world and ideal world are indistinguishable if the UE scheme is IND-ENC secure. In other words, if such a distinguisher exists, then an adversary could use it as a subroutine to break the IND-ENC security of the UE scheme by using the oracles from its game to handle all queries from the distinguisher. Furthermore, as mentioned above, the distinguisher will not force the adversary to lose the game by means of a trivial win, so all queries can be answered. At some point, the distinguisher correctly differentiates between the encryption of a string of zeroes or an actual message (two challenges messages in the IND-ENC game), since this is the only distinctive behavior between both worlds, which tells the adversary how to answer the challenger, and win its game.

C Details of Section 5

The protocol in the bad randomness setting only differs from the basic protocol in the way the owner deals with randomness. As such, the host converter remains unchaged. The owner converter has some minor changes as it incorporates the use of the PRF and PRG, as described in Section 4. The description of the new owner converter is in Fig. 14. To avoid being unnecessarily redundant, we will cover only the changes to Fig. 10.

D Proof of Theorem 2

The proof is structured as follows. First, we introduce a modified IND-ENC-CPA security notion, IND-ENC-CPA*, and prove that it is satisfied by SHINE0. (The notion is quite arbitrary, but it simplifies the proof.) Then, we formally define the simulator and finally prove that the real and ideal worlds are indistinguishable.

D.1 IND-ENC-CPA* Security of SHINE0

IND-ENC-CPA* is the same as IND-ENC-CPA, except the Nxt oracle is replaced by Nxt$^*(r)$ defined in Fig. 15. Intuitively, Nxt$^*(r)$ allows to call ue.nxt with arbitrary randomness r, but marks both new key and token as exposed in case r is given. Similarly, the adversary is allowed to choose randomness for ue.kg executed at the beginning of the experiment, in which case the first key is marked as exposed.

Claim. SHINE0 defined in Fig. 15 is IND-ENC-CPA* secure, in the ideal-cipher model, assuming DDH in G is hard.

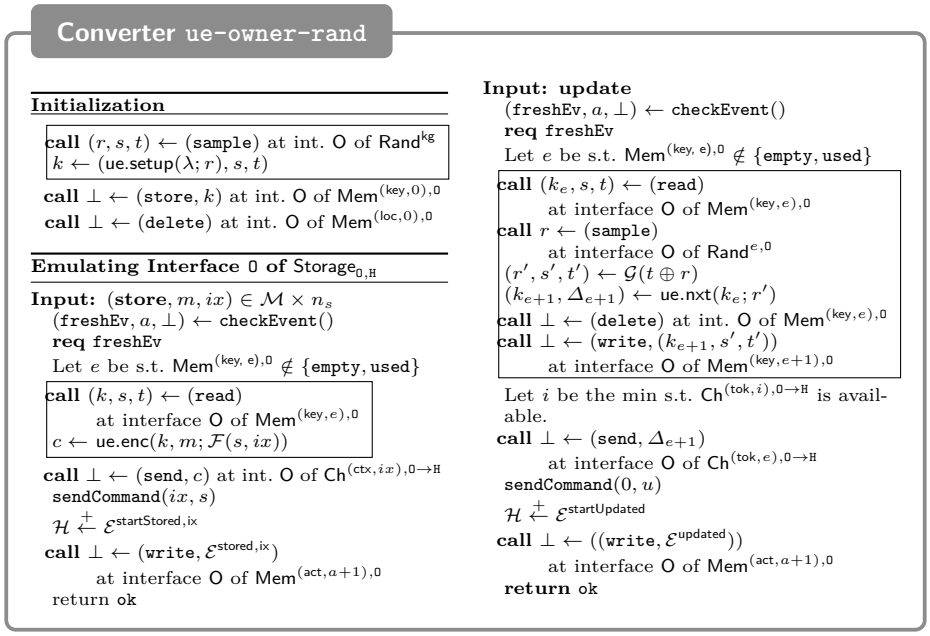

Fig. 14: Changes the owner's converter for the randomness model. The differences from the converter in Appendix B.1 are marked by boxes.

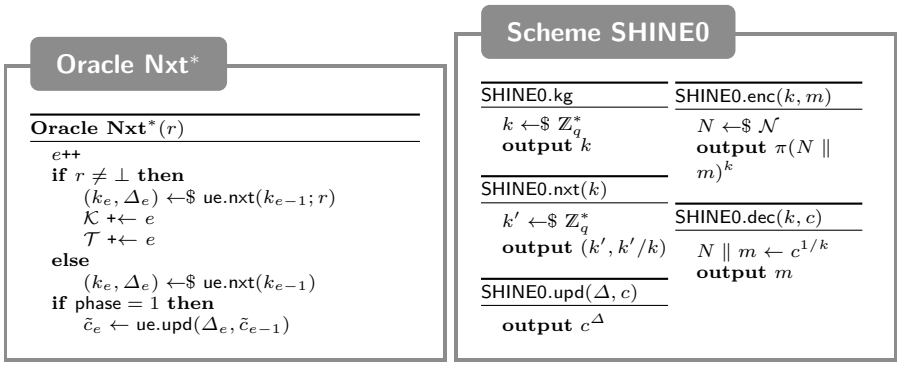

Fig. 15: The modified Nxt* oracle for the IND-ENC-CPA* game and the algorithms of SHINE0.

Proof (Proof of claim). The proof is almost the same as for SHINE0 [BDGJ20]. We sketch it here for completeness. Assume $\mathcal{A}$ is an IND-ENC-CPA* adversary against SHINE0. Given a DDH instance (g^x, g^y, g^z), we emulate $\mathcal{A}$'s oracles and the ideal cipher π as follows; See Fig. 16 for an intuitive picture.

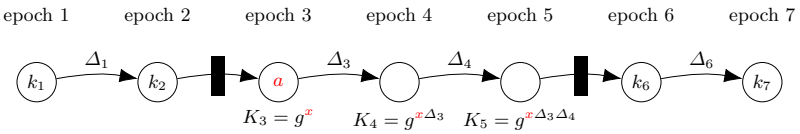

Fig. 16: An example execution with $\mathcal{A}$, where we embed a DDH instance (g^x, g^y, g^z). The firewalls are epochs 2 and 5, marked by ▮. When $\mathcal{A}$ requests challenge in epoch 5, the ciphertext is computed as $(g^z)^{\Delta_3\Delta_4}$ and the ideal cipher is programmed as g^y for m_b and at random for $m_{b\oplus 1}$.

First, we guess three epochs:

1. e^* is the epoch during which $\mathcal{A}$ requests the challenge
2. e_l is the left firewall, i.e. the last epoch before e^* for which the update token is not exposed (technically, exposed means contained in $\mathcal{T}^*$)
3. e_r is the right firewall, i.e., the first epoch after e^* for which the token is not exposed

Then, we keep a (dynamically extended) list of epochs, where each epoch e has a token Δ_e and either a secret key k_e or a public key $K_e = g^{k_e}$. In particular:

– Each epoch except e_l and e_r has a token.
– Each epoch between firewalls (i.e., e s.t. $e_l < e \leq e_r$) has a public key, and each other epoch has a secret key.

The first epoch has a key set during setup (using $\mathcal{A}$'s randomness if necessary). Then, the list is extended when $\mathcal{A}$ queries Nxt as follows. For epochs that are not between firewalls, we can simply set k_e and Δ_e as in Nxt (using $\mathcal{A}$'s randomness if necessary). For the first epoch between firewalls, we set the public key to g^x from the DDH instance. For next epochs between firewalls we choose a random Δ_e (by assumption that the tokens are not exposed, $\mathcal{A}$ must query Nxt with $r = \bot$) and compute the public key using Δ_e and the previous epoch's key.

When $\mathcal{A}$'s queries Enc(m) during an epoch e between firewalls (other Enc queries can be emulated trivially) we compute the ciphertext c follows:

$$
\begin{aligned}
&N \leftarrow\$\ \mathcal{N} \\
&r \leftarrow\$\ |G| \\
&\pi(N \parallel m) \leftarrow\$\ g^r \\
&c \leftarrow K_e^r
\end{aligned}
$$

Note that we program the ideal cipher π at this point. This is possible, because N is random, so $\mathcal{A}$ couldn't have queried π on $N \parallel m$.

To compute the challenge ciphertext c^*, let Δ be the product of all tokens for epochs between e_l and e^* and do:

$$
\begin{aligned}
&N \leftarrow\$\ \mathcal{N} \\
&b \leftarrow\$\ \{0,1\} \\
&\pi(N \parallel m_b) \leftarrow g^y \\
&\pi(N \parallel m_{b\oplus 1}) \leftarrow\$\ G \\
&c^* \leftarrow (g^z)^{\Delta}
\end{aligned}
$$

We output "real" if $\mathcal{A}$ guesses b correctly. If $z = xy$, then c^* is an encryption of m_b. Hence, we perfectly emulated the IND-ENC-CPA* game, which chooses b at random and encrypts m_b. Else, we win with probability 1/2. $\Diamond$

D.2 Simulator

The simulator needs to emulate all `E` interfaces of the randomness resources. Recall that, as long as the key is not compromised, randomness injections have essentially no effect, as guaranteed by the security of $\mathcal{F}$ and $\mathcal{G}$. If, on the other hand, the key is compromised, the adversary is able to decrypt the messages either way, so the effects of bad encryption randomness are not relevant. More concretely, if a ciphertexts leaks before an attempted key leak, then the fix for the commitment problem prevents the key from being leaked to begin with (so, the injections have no effect, and we fall back to rely on the IND-ENC security of the scheme, like in the simple model). If, on the other hand, the key leaks first, then we can just encrypt the same message in both worlds (since the leak predicate of the IW is true). So, it doesn't matter that the ciphertexts are compromised, since the messages are the same. However, we can't simply encrypt the message in the IW with the current key (as we were doing before): even if

both ciphertexts decrypt to the same message, an adversary can very easily tell if a ciphertext was created/updated with good or bad randomness. So, we can't "cheat" and honestly encrypt the message with the current key, since the adversary can trivially tell this apart from an encryption of the same message with bad randomness. That is, we need to dynamically "recreate" the updates. In particular, an adversary could leak all keys and inject all randomness, and the protocol is deterministic for her point of view. So, the strategy oof the simulator will be to use fresh randomness whenever possible, except when a distinguisher would be able to trivially differentiate both world (e.g., a leaking a ciphertext during an exposed epoch), in which case the simulator uses $\mathcal{F}$ and/or $\mathcal{G}$, just like the real world does, in order to preserve consistency.

The formal description of the simulator is very similar to the one in the prior section, $\mathsf{sim}_{\mathtt{UE}}$, which is is described in detail in Fig. 12. We show only the changes to $\mathsf{sim}_{\mathtt{UE}}$. First, besides maintaining a running array of ciphertexts, keys, and tokens, the simulator will also record if the randomness for some epoch change was injected. To do this, we need to update our initialization procedure, and emulate the interface of the new randomness resources. This is shown in Fig. 17.

Lastly, we modify the `updateState()` procedure, to take into account the fact that, in the real world, the encryption and next algorithms are deterministic (and the former nonce-based). That is, as mentioned earlier, fresh randomness will be used for encryptions and updates. However, in situations where the security of $\mathcal{F}$ and/or $\mathcal{G}$ is compromised, the simulator will use $\mathcal{F}$ or $\mathcal{G}$, just like in the real world. For instance, if a ciphertext leaks during an exposed epoch, the randomness parameter of $\mathsf{ue.enc}$ will be computed with $\mathcal{F}$, using part of the secret key as seed. Similarly, if a new key/token needs to be computed during an exposed epoch, and randomness for the epoch change was injected, the randomness parameter of $\mathsf{ue.nxt}$ will be computed with $\mathcal{G}$. The new version of `updateState()` is in Fig. 17.

D.3 Indistinguishability

To prove the indistinguishability between the RW and the IW (assuming that $\mathcal{F}$ is a secure PRF, $\mathcal{G}$ is a secure PRG, and the ue scheme is IND-ENC secure), we will use a hybrid argument. Our worlds are as follows:

- *World 0.* Real world
- *World 1.* The real world, with a slight modification: during an epoch change, if the key hasn't leaked or no randomness was injected, we sample good randomness instead of calling $\mathcal{G}$ (that is, s', t', and r' are sampled randomly). Otherwise, we use the injected randomness and $\mathcal{G}$, setting $(r', s', t') = \mathcal{G}(t \oplus r)$.
- *World 2.* We modify game 1, in order to also get rid of $\mathcal{F}$. During an encryption, if the key hasn't leaked, sample and use good randomness. Otherwise, use the PRF (with either injected or sampled randomness). Note that, during

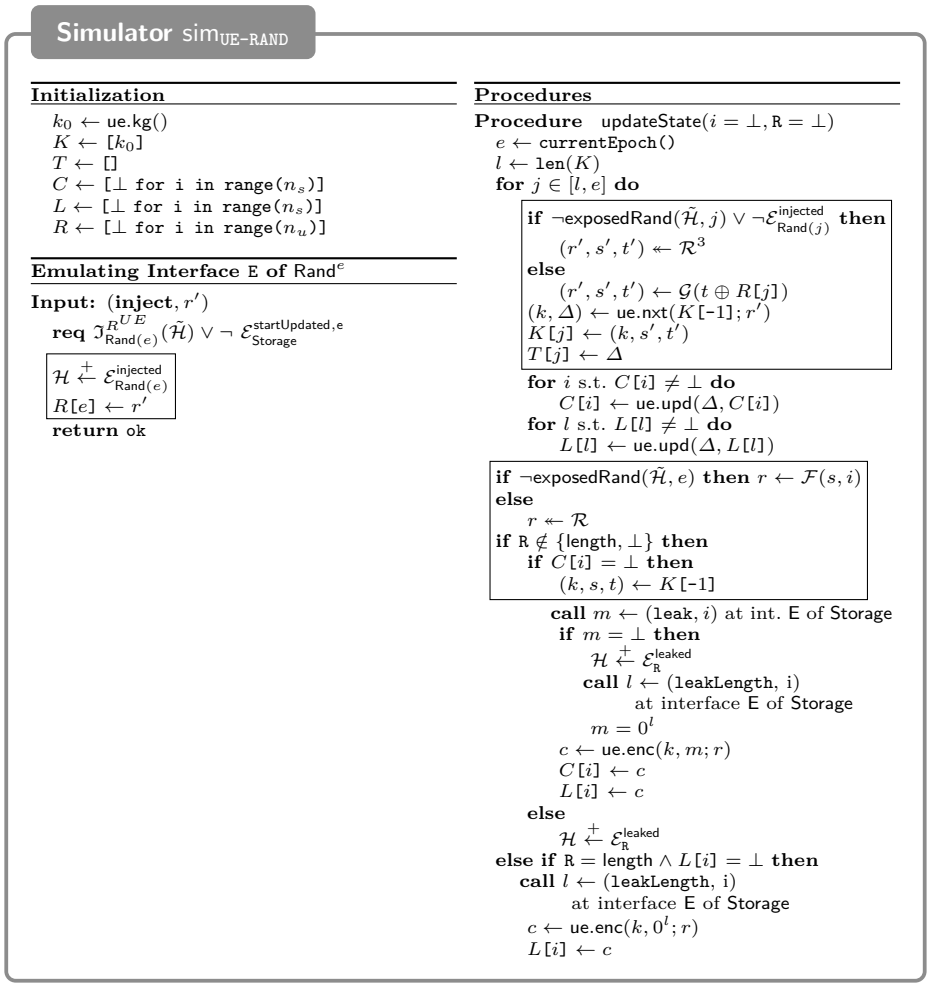

Fig. 17: Changes to the simulator in the randomness model. The differences from the simulator in Appendix B.2 are marked by boxes.

encryptions, the PRF provides no security guarantees if the key/seed leaks, regardless of the randomness used (which makes sense, since the adversary can decrypt either way, regardless of the quality of randomness used).
– *World 3.* Ideal world.

Claim. Worlds 0 and 1 are indistinguishable, assuming that $\mathcal{G}$ is a secure PRG.

Proof (Proof of claim). assume that we have a distinguisher D for worlds 0 and 1. We want to construct an adversary A that breaks the security of the PRG, using D as a subroutine. Recall that A wants to differentiate between $\mathcal{G}(s)$ and r, for randomly sampled s and r. Note that this is precisely the difference between both worlds: if the input is random or the key is secret, we use $\mathcal{G}$ in world 0 and sample a random string in world 1. More formally, A will emulate interfaces $\mathtt{H}, \mathtt{O}$, and $\mathtt{E}$ for D, and execute the real world protocol, with the only difference being how (s, t) are computed. As such, we can ignore the rest of the protocol, and think of A's emulation as specifying a "PRG" chain. In particular, the PRG chain of world 0 is just a sequence of calls to $\mathcal{G}$, and the PRG chain of world 1 is an interleaving of calls to $\mathcal{G}$ and randomly sampled strings (depending on the security of the epoch change). By assumption, D is able to differentiate these two chains (i.e., D exists if and only if he can tell these two apart).

We will now define a series of intermediate chains: the 0-th chain is the chain of world 0, the n_u-th chain is the chain of world 1, and, $\forall i : 0 < i < n_u$, the i-th chain is such that the first $i - 1$ links are from chain n_u, and the rest are from chain 0. It follows then that, assuming said distinguisher exists, he should be able to differentiate (at least) two consecutive chains from each other (if this is not the case, he can't differentiate the endpoints of the sequence of hybrids, due to the triangle inequality). Let i and $i+1$ be two of the consecutive chains that D can distinguish (A can easily find this pair by running D with all the hybrid chains, in order, until D changes his answer). A will now inject the challenge message in place of the i-th link (which is where the two chains differ). Note that this link must correspond to a "good" epoch change (bad epoch changes are treated completely analogously in both worlds (using $\mathcal{G}$ with the leaked key and the bad randomness), so it cannot be possible for D to gain any distinguishing advantage). As such, A now simply runs D and, depending on his answer, replies back to the challenger (if the challenge is a randomly sampled string, this corresponds to chain $i+1$, and if the challenge is the output of $\mathcal{G}$ on a randomly chosen seed, this corresponds to chain i). Importantly, note that the input to $\mathcal{G}$ in chain i will be uniformly random (since either the key is good or the randomness is good, so their XOR is random), which is also the case for the challenge, so D's answer should be consistent with this. Put another way, for whatever the current value of t is, there is some value of r that could be sampled such that $t \oplus r$ equals the challenge seed, for every possible challenge seed.

As a final caveat, note that A never has to reveal his challenge seed. By construction, each link where the challenge is inserted corresponds to a good epoch change. As such, if A has to leak the secret key, this means r was not injected. So, the input to $\mathcal{G}$ is still random, from the adversary's point of view. In

other words, for whatever value of s that D may see, the distribution of possible seeds for $\mathcal{G}$ is still uniform (due to r being uniformly sampled and secret). $\Diamond$

Claim. worlds 1 and 2 are indistinguishable, assuming $\mathcal{F}$ is a secure PRF.

Proof (Proof of claim). Since it is very similar to the above, we will be more brief here. A now tries to differentiate between $\mathcal{F}(s, r)$ and $f(r)$, for randomly chosen f (from the PRF family) and s (from the key space). A and D interact just as explained in the prior proof. In these two worlds, the usage of $\mathcal{G}$ is completely analogous, but the usage of $\mathcal{F}$ is different. So, A specifies a sequence of "PRF chains" instead, which determines if we are in world 1 or 2 (or an intermediate one). Each time $\mathcal{F}$ could be invoked corresponds to a new link (that is, one link for every encryption). The endpoints of the hybrids are the chains from world 1 and world 2, and each intermediate hybrid i is such that the first $i-1$ links are from world 2, and the rest from world 1. By assumption, D is able to differentiate between the two endpoint chains, so he must be able to differentiate between two consecutive chains. If we take any two of said chains, these differ in one link. Note that, as before, this difference needs to correspond to an encryption during an epoch where the key didn't leak (otherwise, the behavior between both worlds is analogous, so D can't gain a distinguishing advantage). However, if the key is still secret, this means that $\mathcal{F}$'s seed is random in world 1. As such, A inserts the challenge in this link, and replies based on D's answer. $\Diamond$

Claim. Worlds 2 and 3 are indistinguishable, assuming that ue is IND-ENC-CPA* secure.

Proof (Proof of claim). The proof that event histories are consistent in both worlds, and that the adversary's outputs are of the same type, is identical to the proof for our first statement Appendix B.2 (the randomness-injected events are trivially consistent). This means that the worlds differ only in that in World 2 all ciphertexts contain the owner's messages, while in World 3, come ciphertexts contain 0's. Now given a distinguisher $\mathcal{D}$, we can construct an adversary $\mathcal{A}$ against IND-ENC-CPA* security of ue as follows: $\mathcal{A}$ executes the code of Storage and the simulator, except arrays K and T are not used. Instead, leakage from token channels and key memories is computed using the Cor oracle. Moreover, the ue algorithms are evaluated as follows:

- Evaluating ue.nxt is replaced by calling the Nxt oracle, with injected randomness if necessary.
- Evaluating ue.upd is replaced by calling the Upd or UpdC oracle.
- To evaluate ue.enc when exposedRand is true, $\mathcal{A}$ corrupts the key and encrypts the message
- To evaluate ue.enc when exposedRand is false, $\mathcal{A}$ calls the Chal oracle with inputs m and 0^l, where m is the message encrypted in World 2 (note that $\mathcal{A}$ knows all messages from the inputs to the owner given by $\mathcal{D}$).

It easily follows by inspection that with our can-leak predicate and the commitment-problem restriction, $\mathcal{A}$ does not trigger the trivial-win flag twf. $\Diamond$

E Proof of Theorem 3

We modify the simulator used in the proof of Theorem 1 in Appendix B.2 as follows. First, when generating an encryption of an unknown message for cell i, instead of encrypting 0's, the simulator encrypts (i, r) for a random message r of appropriate length and stores r in an array $R[i]$. Second, when the distinguisher injects a ciphertext c' into memory $\mathsf{Mem}^{(i,e),\mathtt{H}}$ (and the real-world injection is successful), the simulator decrypts the plaintext ptx and

- If $ptx = (i, R[i])$, it injects original into Storage.
- Else if $ptx = (i, m')$, it injects m' into Storage.
- Else, it injects $\perp$ into Storage.

The intuition is that $\mathcal{D}$ does not know $R[i]$, so any injection descrypting to $R[i]$ is a re-encryption of the ciphertext outputted by the simulator. This holds with high probability, assuming the message space is large. See Fig. 18 for the formal definition of the simulator.

To show that the two worlds are indistinguishable, we consider a sequence of n_s hybrids. In hybrid i, cells 1 to $i-1$ are as in the ideal world, i.e., maintained by Storage and the simulator, and cells i to n_s are as in the real world, i.e., maintained by the protocol and all necessary memories and channels. (The simulator and protocol use the same keys. Upon update, the simulator does not simulate the channels and updates only cells 1 to $i-1$.)

Assume $\mathcal{D}$ distinguishes between hybrids i and $i+1$. We construct an IND-ENC-RCCA adversary $\mathcal{A}$ who emulates $\mathcal{D}$'s experiment as follows. First, it calls Nxt n_u times, so the only actions left are cell-specific.

Cells $j \neq i$ are handled as follows: $\mathcal{A}$ executes the simulator's code (cells 1 to $i-1$) or the protocol's code (cells $i+1$ to n_s). To evaluate ue.enc, ue.upd and ue.dec, call the Enc, Upd and Dec oracles with the message used by the simulator or the protocol. If Dec returns `test`, treat it as $\perp$.

Cell i is handled as follows: Say the message stored by $\mathcal{D}$ in this cell is m. Set $M \leftarrow m$ and when the owner retrieves his message, output M. To create the ciphertext (executed when first leaked), receive challenge c^* on input $ptx_0 = (i, m)$ (owner's message as in hybrid i) and $ptx_1 = (i, r)$ for random r (simulator's message as in hybrid $i+1$). When $\mathcal{D}$ injects c', call the Dec oracle on c', receive ptx and:

- If $ptx = \mathtt{test}$, set $M \leftarrow m$.
- Else if $ptx = (i, m')$, set $M \leftarrow m'$.
- Else, set $M \leftarrow \perp$.

To update, call Upd. All other actions are trivial.

Observe that challenge is of the form (i, m), so if in Step 1. it is injected into cell $j \neq i$, the simulator (decrypts it immediately and) injects $\perp$, and the protocol (decrypts it on retrieve and) outputs $\perp$. Further, note that in Step 1. executing the protocol's code requires calling the Upd oracle with arbitrary ciphertexts injected by $\mathcal{D}$.

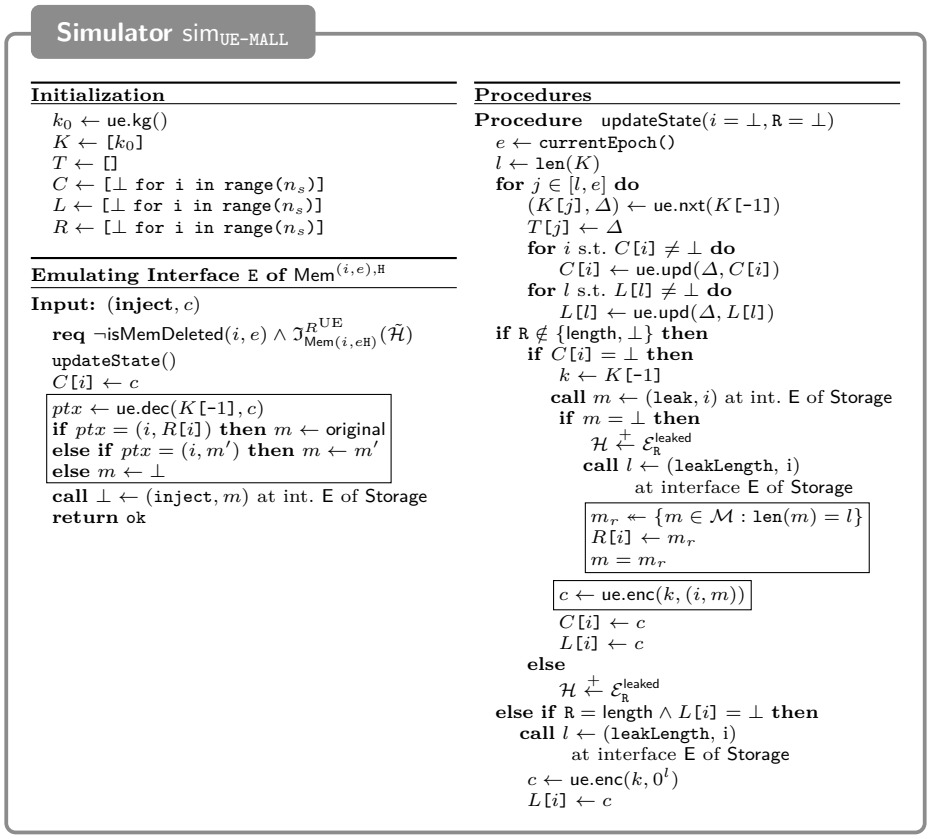

Fig. 18: Changes to the simulator in the malleability model. The differences from the simulator in Appendix B.2 are marked by boxes.

If the challenge c^* contains ptx_1, then $\mathcal{A}$ simulates hybrid $i+1$ exactly (the encrypted message is (i, r) and $\mathcal{A}$ executes the code of the simulator). If c^* contains ptx_0, then emulation is perfect except if $\mathcal{D}$ injects (i, r) (then in $\mathcal{A}$'s emulation the owner retrieves m, and in hybrid i the owner's protocol decrypts and outputs r). However, this happens with negligible probability, because all values seen by $\mathcal{D}$ are independent of r, which is contained only in ptx_1 (assuming that the message space is large).