Compact Key Storage in the Standard Model

Yevgeniy Dodis[⋆] and Daniel Jost

New York University
{dodis, daniel.jost}@cs.nyu.edu

Abstract. In recent work [Crypto'24], Dodis, Jost, and Marcedone introduced Compact Key Storage (CKS) as a modern approach to backup for end-to-end (E2E) secure applications. As most E2E-secure applications rely on a sequence of secrets $(\mathrm{s}_1, \ldots, \mathrm{s}_n)$ from which, together with the ciphertexts sent over the network, all content can be restored, Dodis et al. introduced CKS as a primitive for backing up $(\mathrm{s}_1, \ldots, \mathrm{s}_n)$. The authors provided definitions as well as two practically efficient schemes (with different functionality-efficiency trade-offs). Both, their security definitions and schemes relied however on the random oracle model (ROM).

In this paper, we first show that this reliance is inherent. More concretely, we argue that in the standard model, one cannot have a general CKS instantiation that is applicable to all "CKS-compatible games", as defined by Dodis et al., and realized by their ROM construction. Therefore, one must restrict the notion of CKS-compatible games to allow for standard model CKS instantiations.

We then introduce an alternative standard-model CKS definition that makes concessions in terms of functionality (thereby circumventing the impossibility). More precisely, we specify CKS which does not recover the original secret s_i but a derived key k_i, and then observe that this still suffices for many real-world applications. We instantiate this new notion based on minimal assumptions. For passive security, we provide an instantiation based on one-way functions only. For stronger notions, we additionally need collision-resistant hash functions and dual-PRFs, which we argue to be minimal.

Finally, we provide a modularization of the CKS protocols of Dodis et al. In particular, we present a unified protocol (and proof) for standard-model equivalents for both protocols introduced in the original work.

⋆ Research partially supported by NSF grant CNS-2055578, and gifts from JP Morgan, Protocol Labs, Stellar, and Algorand Foundation.

Table of Contents

1 Introduction . . . 3
  1.1 Contributions . . . 3
  1.2 Outline . . . 5
2 Compact Key Storage . . . 5
  2.1 Overview . . . 5
  2.2 CKS Syntax . . . 6
  2.3 Impossibility of Standard-model CKS . . . 6
  2.4 Weaker Standard Model CKS . . . 8
3 Trapdoor Key Derivation . . . 9
  3.1 Defining TKDFs . . . 9
  3.2 Symmetric TKDF . . . 11
  3.3 A Standard-Model Symmetric-TKDF Construction . . . 12
4 Iterative CKS . . . 13
  4.1 Syntax . . . 14
  4.2 Security . . . 14
  4.3 Constructing I-CKS from TKDF . . . 15
5 CKS from Iterative CKS . . . 18
A Preliminaries . . . 21
B Standard-model CKS: Definitions . . . 21
  B.1 Correctness . . . 21
  B.2 Integrity . . . 23

1 Introduction

Backup is an essential functionality of any application storing user data. For instance, users of a secure messaging (SM) application may expect cloud backup to be provided such that they do not lose their conversation history or sent and received attachments, such as photos, in the event their device is broken, lost, or stolen. The existing cryptographic literature on backup heavily focuses on how to secure a cryptographic secret under a human-memorizable secret, such as a low-entropy password. For instance, WhatsApp combines hardware secure modules (HSM) with PAKE such that users can retrieve a cryptographic secret, that is securely stored on WhatsApp's HSMs, based on their password. Various solutions replacing the trust assumption on the HSM with secret sharing have also been proposed. For example password-protected secret sharing (PPSS) [1,11] and more recent solutions such as updatable oblivious key management [12] and DPaSE [6].

In contrast, very little attention has been paid to *what cryptographic secret* should be securely stored or how this interacts with the security of the application under consideration. Indeed, most end-to-end (E2E) secure applications use the rather naive solution of using a static secret key to symmetrically encrypt the user content and upload it to the cloud. This not only can have determinantal effects on the application's (presumed) security as recently demonstrated by Fábrega et al. [9] but also lacks all of the advanced security properties, such as forward secrecy (FS) and post-compromise security (PCS), we have been accustomed to from E2E-secure protocols. Secure messaging furthermore has a clear push toward enabling large groups with potentially thousands of members — such as the recent IETF Message Layer Security (MLS) standard. The naive backup solution, however, cannot take advantage of this inherent redundancy across users for either storage or bandwidth.

Compact Key Storage. In recent work, Dodis et al. [7] introduce the notion of *compact key storage (CKS)*. Essentially, CKS serves as the backup of underlying secrets of an E2E-secure application, rather than the content itself. For instance, for an SM application using the Double Ratchet protocol, CKS would back up the keys from the symmetric ratcheting layer used to encrypt and authenticate the ciphertexts. In addition, the service provider would then need to retain the original Double Ratchet ciphertexts or outsource them to some cloud storage of the users' choice.

CKS uses a compact secret state that evolves whenever a user's application learns or generates a new key, offloading the storage to an untrusted server. Crucially, (1) any users outsourcing the same sequence of keys can use a shared outsourced storage and (2) CKS allows for fine-grained FS and PCS such that every user can restore exactly the set of keys they once knew and have not erased in the meantime. The compact local state can then be backed up using traditional methods such as HSMs or secret sharing. This has several key benefits:

- **PCS/FS:** When using CKS for backup, the combined application inherits the PCS/FS guarantees of the underlying messaging application. In particular, user can efficiently *erase* messages from their storage and the backup by securely replacing their CKS state with an updated one.
- **Deduplication:** All cloud storage (both the CKS storage and the storage of the application ciphertexts) is shared among all users of a given chat.
- **Delegation:** Fine-grained FS enables efficient delegation of parts of the conversation history. The user can create a copy of their CKS state, erase all parts they wish not to share, and then delegate access by sending the CKS state to another party.

In their work, Dodis et al. observe that the natural security notion — key indistinguishability from randomness, conditioned on the outsourced storage — is impossible. Instead, the authors propose a novel security notion of *preservation security* which, roughly speaking, demands that a broad class of applications remains secure when enhanced by CKS. That is, for each given application one needs to show that it is "CKS compatible" to deduce that it can be securely augmented by any secure CKS scheme. They then provide formal definitions of preservation security and present efficient schemes. However, both the security notion and the protocol inherently live in the Random Oracle model (ROM).

1.1 Contributions

In this work, we investigate CKS in the standard model.

Impossibility of preservation security. First, we show that the notion of preservation security by Dodis et al. inherently requires an idealized model. More concretely, we show that for any CKS scheme in the standard model, there exists a CKS-compatible game that becomes insecure when enhanced with the CKS scheme.[1]

Standard model CKS definitions. While the original CKS notion was aimed at augmenting any (legacy) E2E-secure application, we observe that if the application is designed with CKS in mind, then the aforementioned impossibility can be circumvented. More concretely, we observe the following: if instead of recovering the original secrets $(\mathrm{s}_1, \ldots, \mathrm{s}_n)$, henceforth called *seed*, each party only needs to recover a derived *key* $(\mathrm{k}_1, \ldots, \mathrm{k}_n)$ then the impossibility no longer holds. For instance, assume that the seeds are

[1] Note the order of quantifiers. A stronger statement that there exist CKS-compatible applications that are insecure for any CKS scheme is conceivable, but left to future work. Still, our result means we cannot instantiate the definition from [7].

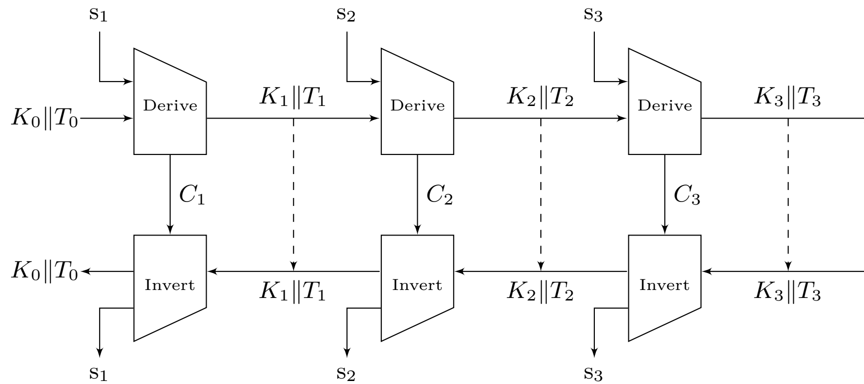


Fig. 1: [7, Fig. 7]. A schematic representation of all-or-nothing CKS scheme from [7]. The top half depicts the parties outsourcing the keys, while the lower half shows the key-recovery process.

the output of a Continuous Group Agreement (CKA) or Continuous Group Key Agreement (CGKA) protocol. Our proposed notion is compatible with any E2E-secure application that only uses a derived key $\mathrm{k}_i = \mathrm{KDF}(\mathrm{s}_i)$ instead of the seed. This can either be a new application explicitly with CKS in mind (in which case the standard-model CKA scheme gets to choose the KDF) or a legacy application that already happens to involve an additional key derivation step (under modest assumptions on the KDF).

On a high level, we observe that the impossibility stems from circularity that necessitates an idealized model such as the ROM. Having an explicit key derivation step then resolves this circularity. Indeed, we observe that for this weaker notion of CKA, indistinguishability from randomness is achievable, no longer necessitating the (rather intricate) notion of preservation security. We then adapt the CKS notion accordingly. Our standard-model CKS is incomparable with the original ROM-based definition from [7]. On the one hand, we obtain the stronger indistinguishability-based notion. On the other hand, we have to concede in several aspects:

- As mentioned above, standard-model CKS recovers only keys instead of the seeds. This makes it (potentially) unsuitable for some legacy applications.
- Delegation of keys only. Similarly, parties can only delegate keys. This somewhat restricts functionality as the receiving party can no longer equally contribute to the shared outsourced state without the original seeds.
- Selective security only. Fully adaptive security seems to imply non-committing encryption for messages longer than the key, as the compact local state is significantly shorter than the total length of the keys that are outsourced. This is generally known to be impossible in the standard model. We therefore settle for selective security, even though for (some of) our schemes the framework by Kamath et al. [13] should yield at least quasi-polynomial reductions against adaptive adversaries.

Standard-model CKS schemes. We present an efficient scheme for standard-model CKA. When targeting outsider security and an honest-but-curious server, our scheme only needs a PRG and one-time secure symmetric encryption. In other words, in its weakest form, it can be constructed purely from one-way functions. When targeting an actively malicious server, we additionally need collision-resistant hash functions. To achieve insider security, where either parties delegate inconsistent keys, or already start with inconsistent seeds, our scheme additionally needs a Dual-PRF [2,4], for which it is an open problem whether they can be constructed from one-way functions.

Modularization. Dodis et al. [7] present two CKS schemes: One which allows all-or-nothing delegation (and all-or-nothing erasure) and which has a constant size local state $\mathrm{st_u}$. The other allows to efficiently delegate any continuous interval of secrets and has a local state that grows logarithmically in the number of epochs. At their core, both schemes use Convergent Encryption (CE) [8] to recursively aggregate the two secrets into one and a ciphertext. Slightly simplified, the former scheme aggregates the old state $\mathrm{st_u}$ and the secret s for the next consecutive epoch as:

- Parse $(K, T) \leftarrow \mathrm{st_u}$
- Compute a new key $K' \leftarrow \mathrm{H}(\mathrm{st_u}||\mathrm{s})$
- Compute $C' \leftarrow \mathrm{SE.Enc}(K, (\mathrm{st_u}||\mathrm{s}))$
- Compute $T' \leftarrow \mathrm{H}(C'\|T)$
- Set $\mathrm{st}'_\mathrm{u} \leftarrow (K, T)$ and send $\mathrm{C} \leftarrow (C, T')$ to the server.

This process is called "derive" in the schematic representation of the all-or-nothing scheme in Fig. 1. The security of the scheme inherently requires the ROM for the key generation in the second step. Moreover, the authors of [7] observe that the abstraction of CE as Message Locked Encryption (MLE) by Bellare et al. [3] does not apply to the recursive application, and the authors instead proved the scheme's security directly based on the security of the symmetric encryption SE. Enc and the ROM. This raises the question: what is the appropriate substitute for the above "derivation box" for standard-model CKS? Observe that while the above construction allows to recover s based on the ciphertext and the local state, whereas standard-model CKS only needs to recover a key that is derived

from the seed. We, therefore, answer this question by introducing the abstraction of a *Trapdoor KDF (TKDF)*. A TKDF, in a nutshell, represents a kind of "invertible" KDF that generates a key k and updated state z' from a seed s and previous state z, such that from z a secret *trapdoor* t to invert the operation can be derived.

As a second abstraction, we introduce the notion of *iterative CKS*, which is a class of CKS protocols that encompass both schemes from [7]. This abstraction has several benefits: (1) its definitions are significantly simpler than the ones of fully general CKS; (2) it allows us to formalize and prove the security of a natural unified protocol of which the all-or-nothing and the interval protocol are special cases. In particular, we note that the iterative CKS notion reduces finding the right trade-off between functionality (in terms of delegation and fine-grained erasure) and efficiency of the scheme to a graph theoretic problem. The class of graphs involved is further closely related to graphs studied in pebbling games, for instance, allowing us to link it with the work on adaptive security by Jafargholi et al. [10] and Kamath et al. [13].

1.2 Outline

In Section 2, we first argue the impossibility of standard-model CKS, according to the definition of [7], and then introduce an alternative definition with weakened functionality (which still suffices for many practical applications). In Section 3, we then introduce the TKDF notion, abstracting over the core component of both constructions from [7], and provide an efficient instantiation from standard-model primitives. In Section 4 we introduce a special case of CKS protocols, dubbed iterative CKS, and show how a generic protocol (based on TKDF) abstracts over both protocols from [7]. Finally, in Section 5, we sketch how any iterative CKS protocol implies a general (standard model) CKS protocol. Some preliminaries are presented in Appendix A.

2 Compact Key Storage

2.1 Overview

Recall from Section 1 that Compact Key Storage allows a group of users that know a shared set of secrets $(\mathrm{s}_1, \ldots, \mathrm{s}_n)$ to maintain a shared backup of those secrets. To this end, for every new secret s_i obtained by the users, *at least one* user uploads a ciphertext C_i to the (untrusted) server. Each user u only keeps a small local state $\mathrm{st_u}$ and the server storage should not grow in the number of users — hence the name *compact*. All users must then be able to use their local state and the ciphertexts to recover all secrets they once knew (and have not explicitly erased). Importantly, [7] introduced CKS for *dynamic* groups, meaning that not every user necessarily knows all secrets. On the contrary, a user who does not know a certain secret s_i should not be able to derive any information about s_i from the CKS ciphertexts. This, in turn, implies post-compromise security, as $\mathrm{st_u}$ before the user u learned s_i together with all ciphertexts $(\mathrm{C}_1, \ldots, \mathrm{C}_n)$ must not leak information about s_i. Unfortunately, it was shown in [7] that the natural definition of s_i remaining pseudorandom given all ciphertexts $(\mathrm{C}_1, \ldots, \mathrm{C}_n)$, and a party's initial state, is impossible even in idealized models.

Before recapping the formal CKS notion of [7], let us provide a high-level summary of the desired functionality.

- *Outsourcing secrets.* Each user can append a secret s_i for a new epoch i to their local state. To save bandwidth, this should, intuitively, be an operation done locally by all users except one. As this is infeasible when having users who know substantially different subsets of secrets, [7] relaxed this condition slightly. More precisely, a user U should upload unless another user U', who knew a superset of the secrets of the former, did the upload when learning s_i.
- *Retrieving secrets.* Whenever the user U later wants to retrieve one or more of the secrets they knew at some point, they should be able to use their local state $\mathrm{st_u}$ and the help of the server. The notion of [7] imposes *integrity* to ensure that U will never retrieve a different secret than they originally knew. Of course, as for any outsourced storage scheme a malicious server can do a denial-of-service attack. A strong correctness notion, however, ensures that as long as the server is honest other malicious group members cannot prevent U from retrieving their secrets.
- *Key delegation.* A user U should be able to delegate access to the entirety or parts of their secrets by sending a short message msg over a secure channel to any other user U'. Using msg, U' should then be able to recover those secrets with the help of the server.
- *Key erasure.* To securely delete specific content of the application, the user U should be able to delete access to their secrets. In other words, U may wish to erase s_i (or more generally a subset of the secrets) such that afterward their updated state $\mathrm{st_U}$ and the ciphertexts $(\mathrm{C}_1, \ldots, \mathrm{C}_n)$ no longer reveal information about s_i, without requiring the server to securely erase information. This can be seen as a special case of delegation where the user delegates themselves the set of secrets they wish to retain; more efficient implementations are however conceivable.

It is not too hard to see that it is impossible for a scheme to support erasing and delegating arbitrary subsets of the secrets while maintaining a compact local state. Thus, [7] parametrized each concrete CKS in the set of operations that it supports efficiently. Those sets are described as predicates that determine whether an operation is feasible for a set of epochs share given the set of epochs know for which the user currently "knows" the secrets (i.e., learned and not erased them).

Definition 1 ([7, Def. 1]). *A* delegation family $\mathcal{G}$ *is a predicate* $\mathcal{G}\colon \mathcal{P}(\mathbb{N}) \times \mathcal{P}(\mathbb{N}) \to \{0,1\}$*, where for a set* $\text{know} \subseteq \mathbb{N}$ *of epochs with the respective keys known to a party,* $\mathcal{G}(\text{know}, \text{share})$ *indicates whether they can delegate* $\text{share} \subseteq \mathbb{N}$*. Analogously, a* retrieval family $\mathcal{R}$ *and a* erasure family $\mathcal{E}$ *indicate whether the party can recover* $\text{share} \subseteq \mathbb{N}$ *or erase* $\text{share} \subseteq \mathbb{N}$*, respectively.*

2.2 CKS Syntax

We now recap the Compact Key Storage notion. The following section is mostly taken verbatim from [7].

Definition 2 ([7, Def. 2]). *A* Compact Key Storage (CKS) *scheme* CKS *for a delegation family* $\mathcal{G}$*, a retrieval family* $\mathcal{R}$*, and an erasure family* $\mathcal{E}$ *(or* $(\mathcal{G},\mathcal{R},\mathcal{E})$*-CKS for short) is an interactive protocol between stateful user* U *and server* S *algorithms, respectively, defined by the following sub-algorithms:*

Initialization:

- *The* $\text{st}_\text{S} \leftarrow \text{S.Init}(1^\kappa)$ *algorithm initializes the server's state.*
- *The* $\text{st} \leftarrow \text{U.Init}(1^\kappa)$ *algorithm initializes a user's state.*

Key Management:

- *The non-interactive append algorithm takes the current state* st*, an epoch* $\mathtt{e}$*, a secret* s*, and flag* upload*. The invocation*

$$(\text{st}', \text{st}_\text{up}) \leftarrow \text{U.Append}(\text{st}, \mathtt{e}, \text{s}, \text{upload}),$$

  produces an updated state st′ *and, if* $\text{upload} = \texttt{true}$*, an upload state* st_up*. (If* $\text{upload} = \texttt{false}$*, then* $\text{st}_\text{up} = \bot$*.)*
- *The interactive upload algorithm takes the upload state and after the interaction*

$$(\bot; \text{st}'_\text{S}) \leftarrow \langle \text{U.Upload}(\text{st}_\text{up}) \leftrightarrow \text{S.Upload}(\text{st}_\text{S}) \rangle,$$

  the server outputs an updated state st'_S*.*
- *The interactive erase algorithm takes the current state* st *and a set of epochs* $\text{share} \subseteq \mathbb{N}$*. After the following interaction*

$$(\text{st}'; \bot) \leftarrow \langle \text{U.Erase}(\text{st}, \text{share}) \leftrightarrow \text{S.Erase}(\text{st}_\text{S}) \rangle,$$

  both the user outputs an updated state st′ *(and the server has no output).*

Delegation:

- *The interactive granting algorithm takes a user* U_1*'s state* st_1 *and a set* $\text{share} \subseteq \mathbb{N}$ *of keys to be shared with another user* U_2*. After the interaction*

$$(\text{msg}; \bot) \leftarrow \langle \text{U}_1.\text{Grant}(\text{st}_1, \text{share}) \leftrightarrow \text{S.Grant}(\text{st}_\text{S}) \rangle$$

  the user outputs the information msg *to be sent to the other party* U_2*.*
- *The interactive grant-accepting algorithm extends another user's* U_2 *known key set by processing a grant* msg*. After the interaction*

$$(\text{st}'_2, \text{st}_\text{up}; \bot) \leftarrow \langle \text{U}_2.\text{Accept}(\text{st}_2, \text{share}, \text{msg}, \text{upload}) \leftrightarrow \text{S.Accept}(\text{st}_\text{S}) \rangle$$

  the user outputs an updated state st'_2*, as well as (if* $\text{upload} = \texttt{true}$*) a state for the* Upload *algorithm.*

Retrieval:

- *The interactive key-retrieval algorithm restores the secrets for epochs* $\text{share} \subseteq \mathbb{N}$ *with the interaction*

$$(\text{secrets}; \text{st}'_\text{S}) \leftarrow \langle \text{U.Retrieve}(\text{st}, \text{share}) \leftrightarrow \text{S.Retrieve}(\text{st}_\text{S}) \rangle$$

  ending with the user outputting a function $\text{secrets}\colon \text{share} \to \text{s}$*, as well as an updated server state.*

A CKS scheme is considered efficient if all operations work in sublinear — ideally logarithmic — time in the number of epochs n (when secrets are appended in consecutive order). As such, the predicates $(\mathcal{G},\mathcal{R},\mathcal{E})$ dictate efficiency requirements: if for instance a party wants to retrieve an arbitrary set $\mathcal{I}$ of epochs, they can find a minimal cover $\mathcal{I} = \mathcal{I}_1 \cup \cdots \cup \mathcal{I}_k$ of subsets consistent with $\mathcal{R}$ and retrieve subset. This leads to an overall efficiency of $\mathcal{O}(k \log(n))$. In terms of client state, we require it to grow at most in the order of $\mathcal{O}(d \log(n))$, with d denoting the number of erasure operations.

In case secrets are appended sparsely (such as odd epochs only), are appended completely out of order, or linearly many erasures have been performed, efficiency may degrade to linear time. The server state must grow at most linearly in the number of overall epochs outsourced by any party, and in particular, must not grow in the number of participating parties.

2.3 Impossibility of Standard-model CKS

The authors of [7] showed that any non-trivial state compactness guarantee of the user makes it impossible to satisfy the most desirable key indistinguishability property for CKS, even if one only wants to recover all n secrets $(\text{s}_1, \ldots, \text{s}_n)$ from the latest state st_n of the user, with the help of the CKS

server. Concretely, the following probability cannot be upper bounded by $\frac{1}{2} + \mathsf{negl}(\kappa)$ for all efficient attackers $\mathcal{A}$:

$$\Pr\left[b = b' \;\middle|\; \begin{array}{r} b \leftarrow\!\$ \{0,1\} \\ \mathrm{st} \leftarrow \mathrm{U.Init}(1^\kappa); \mathrm{st_S} \leftarrow \mathrm{S.Init}(1^\kappa) \\ \mathrm{s}_1^0, \mathrm{s}_1^1, \mathrm{s}_2^0, \mathrm{s}_2^1, \ldots, \mathrm{s}_n^0, \mathrm{s}_n^1 \leftarrow\!\$ \{0,1\}^\kappa \\ \forall i \in [n] : (\mathrm{st}, \mathrm{st_{up}}) \leftarrow \mathrm{U.Append}(\mathrm{st}, i, \mathrm{s}_i^0, \mathtt{true}), \\ (\bot; \mathrm{st_S}) \leftarrow \langle \mathrm{U.Upload}(\mathrm{st_{up}}) \leftrightarrow \mathrm{S.Upload}(\mathrm{st_S}) \rangle \\ b' \leftarrow \mathcal{A}(1^\kappa, \mathrm{s}_1^b, \ldots, \mathrm{s}_n^b, \mathrm{st_S}) \end{array} \right]$$

Intuitively, one cannot expect that the n secrets $(\mathrm{s}_1, \ldots, \mathrm{s}_n)$ are still pseudorandom when the attacker gets access to the CKS functionality (in particular, server public storage $\mathrm{st_S}$). This is because CKS provides a testable functionality — (user) state compaction — which is not possible with random unrelated secrets.

To circumvent this result in the random oracle model (ROM), the authors introduced a rather intricate, weaker notion of "CKS-preservation", described below. Note that while the authors of [7] observed that relying on an idealized model for CKS-preservation seemed inherent, no formal result was proven. In the next section, we close this gap, showing that preservation security is impossible *in the standard model*. Since preservation security was meant to be the weakest meaningful security notion for CKS this, in spirit, establishes that CKS recovering the original secrets (as a standalone primitive) is impossible in the standard model.[2]

Impossibility of CKS-preservation. Intuitively, CKS-preservation relaxes key indistinguishability with the requirement that access to the CKS functionality does not hurt the security of the underlying application Π (originally not designed with CKS in mind). To make this statement non-tautologous, [7] had to define the types of applications Π where this makes sense, without using CKS-syntax inside Π, but still keeping Π as general as possible. They call such games *CKS-compatible*. Below we give a special case of such a CKS-compatible game, which already shows the impossibility of standard-model CKS-presevation.

Concretely, we will concentrate on the (subset of) CKS-compatible games Π where: (1) Π has a sequence of secrets $(\mathrm{s}_1, \ldots, \mathrm{s}_n)$; (2) Π remains secure even if permits the adversary $\mathcal{A}$ has access to the testing oracle $\mathsf{Test}(i, \mathrm{s})$ which returns 1 if $\mathrm{s} = \mathrm{s}_i$. The ROM-based construction of [7] worked for all the games in this class, provided the honest parties do not use the random oracle utilized by the CKS.[3] Thus, to show the standard-model impossibility of instantiating this result, we only need to construct a single game Π which satisfies properties (1) and (2), but where the knowledge of server state $\mathrm{st_S}$ will break Π.

Counter-Example Game. We consider the following game Π between the challenger $\mathcal{C}$ and the polynomial-time attacker $\mathcal{A}$, where κ is the security parameter, and n is chosen large enough so that the length of the CKS state st_n after appending n random secrets $\mathrm{s}_i \in \{0,1\}^\kappa$ satisfies $|\mathrm{st}_n| \leq n(\kappa - \omega(\log \kappa))$.

1. $\mathcal{C}$ samples random $\mathrm{s}_1, \ldots, \mathrm{s}_n \in \{0,1\}^\kappa$.
2. $\mathcal{C}$ computes the states of the user after appending $\mathrm{s}_1, \ldots, \mathrm{s}_n$:

$$\mathrm{st}_0 \leftarrow \mathrm{U.Init}(1^\kappa)$$
$$\forall i \in [n] : (\mathrm{st}_i, \cdot) \leftarrow \mathrm{U.Append}(\mathrm{st}_{i-1}, i, \mathrm{s}_i, \mathtt{false}),$$

3. $\mathcal{C}$ sends st_n to $\mathcal{A}$.
4. $\mathcal{C}$ honestly responds to $\mathsf{Test}(i, s)$ queries of $\mathcal{A}$: return 1 iff $s = s_i$.
5. $\mathcal{A}$ send guess values $\mathrm{s}'_1, \ldots, \mathrm{s}'_n$.
6. $\mathcal{C}$ outputs 1 iff $\forall i \in [n] \;\; \mathrm{s}_i = \mathrm{s}'_i$.

First, we argue that this game is easily won by the attacker if the attacker $\mathcal{A}$ *additionally* gets the server state $\mathrm{st_S}$ when the standard-model CKS is applied to $\mathrm{s}_1, \ldots, \mathrm{s}_n$, as allowed by the CKS-preservation security security definition from [7]. This is true because the correctness of the CKS holds even when the Append and the Upload algorithms are run by the different users. Namely, the adversary can still successfully recover the original secrets $\mathrm{s}_1, \ldots, \mathrm{s}_n$, by using the "Alice's state" st_n (given to $\mathcal{A}$ by the challenger $\mathcal{C}$ above) and the server state $\mathrm{st_S}$ obtained when "another user Bob" (corresponding to the helper in the "CKS-enhanced game" of [7]) uploaded the corresponding ciphertexts to the server. Hence, CKS-preservation does not happen for Π.

Second, we nevertheless argue that the original game Π is "CKS-compatible". Namely, Π is secure against any polynomial time attacker $\mathcal{A}$, who does not get to see the server state st_S in Π, but is allowed to have the Test oracle. To see this, we use a standard compression argument. Let us say that $\mathcal{A}$ made $q = \mathsf{poly}(\kappa)$ queries to the Test oracle. The key observation is to notice that one can compactly encode all q responses using significantly fewer than q bits. Namely, for each $i \in [n]$, we only need to know the first index $j \in [q+1]$ where the q's query of $\mathcal{A}$ was successful (or set $j = q + 1$ if this never

[2] In the standard model, our result is a strict strengthening of the impossibility result of key-indistinguishable CKS from [7], but does not generalize to idealized models.
[3] In practice, this is easy to accomplish with salting.

happened). Thus, all q answers obtained by $\mathcal{A}$ take at most $n\log(q+1) = O(n\log\kappa)$ bits to encode. Now, if the attacker wins Π with non-negligible probability, we can use $\mathcal{A}$ to successfully compress $n\kappa$ truly random bits $\mathrm{s}_1, \ldots, \mathrm{s}_n$ as follows:

– Include the final state st_n, whose size is assumed to be $|\mathrm{st}_n| \leq n(\kappa - \omega(\log\kappa))$.
– Include the encoding of q test queries of $\mathcal{A}$, which takes at most $n\log(q+1) = O(n\log\kappa)$ bits to encode.

Combined, this information is enough to run the attacker to produce its guess $\mathrm{s}'_1, \ldots, \mathrm{s}'_n$. Yet, the compression string has overall length $|\mathrm{st}_n| + n\log(q+1) \ll n\kappa$. Which means that the probability that $(\mathrm{s}'_1, \ldots, \mathrm{s}'_n) = (\mathrm{s}_1, \ldots, \mathrm{s}_n)$ must be negligible. Which shows that game Π is CKS-compatible.

Discussion. We make several observations. First, our game Π was allowed to depend on the (hypothetical) algorithms of the standard-model CKS. Indeed, unlike ROM, there is no effective mechanism to prevent honest users from (artificially) utilizing the CKS inside the game Π, if Π is only restricted to satisfy very weak properties (1) and (2) for CKS-preservation.[4] Notice, that simple techniques like utilizing the common reference string do not help, since that string should be available to the users to run the CKS, and a general application (even CKS-compatible) could still "trick" the users to use the right common reference string. Second, our counter-example above is extremely artificial, specifically targeting to break CKS-compatibility, while supporting the Test oracle. This is expected, since for most "natural" applications, such as Signal or MLS, we expect the heuristic instantiation of the ROM-based construction of [7] to be secure in the real world. Instead, the counter-example below should be viewed from the lens that "CKS-preservation" does not appear to be the right security notion of CKS for *standard-model* instantiations. Indeed, our standard model solution will go back to the clean and elegant key-indistinguishability, but will slightly change the functionality of the application to circumvent the impossibility result below.

2.4 Weaker Standard Model CKS

In this section, we now introduce our new notion of standard-model CKS. Simply put, we distinguish between *seeds* $(\mathrm{s}_1, \ldots, \mathrm{s}_n)$ and *keys* $(\mathrm{k}_1, \ldots, \mathrm{k}_n)$, where each key is (deterministically) derived from its respective seed. More concretely, U.Append appends a seed s_i whilst U.Retrieve later recovers the respective key k_i. In addition, delegation is assumed to only delegate keys rather than seeds. This weaker notion of CKS, of course, is only compatible with applications that distribute seeds, for instance as part of a CGKA, but then use keys for the message encryption layer.

Definition 3. *A* Standard Model Compact Key Storage (CKS) *scheme* CKS *is a CKS scheme for which there is additionally a deterministic algorithm* U.Key$(\mathsf{e}, \mathrm{s}) \to \mathrm{k}$ *which takes an epoch* e *and a seed* s*, and returns the corresponding key* k*. A standard-model CKS scheme is defined with respect to a generalized delegation family* $\mathcal{G}$*, retrieval family* $\mathcal{R}$*, and erasure family* $\mathcal{E}$*, each of which takes two arguments: the set of epochs for which a party knows the seeds (and therefore also the keys) and the set of epochs for which they know the keys only. Finally, for consistency, we denote the output of* U.Retrieve *as* keys *(instead of* secrets*).*

Correctness and Security. We now adapt correctness and security to the standard-model setting. The former remains mostly unchanged from ROM-CKS with the obvious difference that U.Retrieve now must return the correct keys instead of seeds. That is, if a user for epoch e appended a seed s to their CKS state, then correctness requires that U.Retrieve later recovers U.Key(e, s), instead of s as in the ROM-CKS notion.

ROM-CKS formalized security as two properties: preservation security and integrity. Intuitively, the former demands that applying CKS to a so-called "CKS-compatible" application does not undermine that application's security. The latter, on the other hand, requires that the CKS scheme only recovers correct seeds — that is, the ones they initially appended to their state — or an error, for an honest party interacting with a malicious server potentially colluding with other malicious users. Integrity of standard-model CKS also remains mostly unchanged from ROM-CKS, with the obvious changes to accommodate the key derivation. For concreteness, we present both the adapted correctness and integrity games in Appendix B.

In the remainder of this section, we present the key-indistinguishability notion of standard-model CKS. This notion replaced the preservation-security notion and, intuitively, represents the desired best-possible security.

Definition 4. *We say that a standard-model* $(\mathcal{G}, \mathcal{R}, \mathcal{E})$*-CKS scheme* CKS *is* key indistinguishable*, if the probability of any PPT adversary* $\mathcal{A}$ *winning the* $(\mathcal{G}, \mathcal{R}, \mathcal{E})$-CKS-$\mathsf{KeyIndist}^{\mathcal{A}}_{\mathsf{CKS}}$ *game from Fig. 2 is negligible in* κ*.*

The goal of the adversary is to guess whether the game uses real keys ($b = 0$) or random ones ($b = 1$). The game follows the template of the preservation-security game from [7], which is very similar to the one of the correctness and integrity games. Notably, the adversary has various oracles mirroring the CKS algorithms. For each of the interactive algorithms, the adversary furthermore assumes the role of the server; that is, the game considers an actively malicious server. (Note that we assume the adversary

[4] Indeed, we will later observe (cf. Theorem 3 and Corollary 4) that for a special sub-class of protocols, we can build provably secure CKS in the standard model, even satisfying key indistinguishability!

Game $(\mathcal{G}, \mathcal{R}, \mathcal{E})$-CKS-KeyIndist$_{\mathsf{CKS}}^{\mathcal{A}}$ (Security)

Main

$b \leftarrow\$ \{0,1\}$
$\mathrm{n} \leftarrow 0$
$\mathsf{St}[\cdot], \mathsf{Secret}[\cdot], \mathsf{Keys}[\cdot], \mathsf{GrantInfo}[\cdot] \leftarrow \bot$
$\mathsf{KnownSeed}[\cdot,\cdot], \mathsf{KnownKey}[\cdot,\cdot], \mathrm{ActualKey}[\cdot,\cdot] \leftarrow \texttt{false}$
$(\mathrm{st}_{\mathcal{A}}, \mathit{Corr}) \leftarrow \mathcal{A}(1^\kappa)$
$b' \leftarrow \mathcal{A}^{\text{CreateUser},\dots,\text{Accept}}(1^\kappa, \mathrm{st}_{\mathcal{A}})$
return $b' = b$

Oracle CreateUser

$n \leftarrow n + 1$
$\mathsf{St}[\mathrm{n}] \leftarrow \mathrm{U.Init}(1^\kappa)$

Oracle Corrupt

Input: $\mathrm{u} \in [\mathrm{n}]$
req $\mathrm{ActualKey}[\mathrm{u},\cdot] \subseteq \mathsf{Corr}$
return $\mathsf{St}[\mathrm{u}]$

Oracle Append

Input: $(\mathrm{u}, \mathsf{e}, \mathrm{s}) \in [\mathrm{n}] \times \mathbb{N} \times \{0,1\}^\kappa$
req $\neg\mathsf{KnownSeed}[\mathrm{u}, \mathsf{e}]$
$(\mathrm{s}, \mathrm{k}) \leftarrow \texttt{sample-if-nec}(\mathrm{u}, \mathsf{e}, \mathrm{s})$
$\mathsf{KnownKey}[\mathrm{u}, \mathsf{e}] \leftarrow \texttt{true}$;
$\mathsf{KnownSeed}[\mathrm{u}, \mathsf{e}] \leftarrow \texttt{true}$
try $(\mathsf{St}[\mathrm{u}], \mathrm{st_{up}}) \leftarrow \mathrm{U.Append}(\mathsf{St}[\mathrm{u}], \mathsf{e}, \mathrm{s}, \mathrm{upload})$
try $\langle \mathrm{U.Upload}(\mathrm{st_{up}}) \leftrightarrow \mathcal{A} \rangle$
return k

Oracle Erase

Input: $(\mathrm{u}, \mathrm{share}) \in [\mathrm{n}] \times \mathcal{P}(\mathbb{N})$
req $\mathcal{E}(\mathsf{KnownSeed}[\mathrm{u},\cdot], \mathsf{KnownKey}[\mathrm{u},\cdot], \mathrm{share})$
try $(\mathsf{St}[\mathrm{u}]; \bot) \leftarrow \langle \mathrm{U.Erase}(\mathsf{St}[\mathrm{u}], \mathrm{share}) \leftrightarrow \mathcal{A} \rangle$
for $\mathsf{e} \in \mathrm{share}$ **do**
  $\mathsf{KnownSeed}[\mathrm{u}, \mathsf{e}] \leftarrow \texttt{false}$; $\mathsf{KnownKey}[\mathrm{u}, \mathsf{e}] \leftarrow \texttt{false}$
  $\mathrm{ActualKey}[\mathrm{u}, \mathsf{e}] \leftarrow \texttt{false}$

Oracle Retrieve

Input: $(\mathrm{u}, \mathrm{share}) \in [\mathrm{n}] \times \mathcal{P}(\mathbb{N})$
req $\mathcal{R}(\mathsf{KnownSeed}[\mathrm{u},\cdot], \mathsf{KnownKey}[\mathrm{u},\cdot], \mathrm{share})$
try $(\mathrm{keys}; \bot) \leftarrow \langle \mathrm{U.Retrieve}(\mathsf{St}[\mathrm{u}], \mathrm{share}) \leftrightarrow \mathcal{A} \rangle$
for $\mathsf{e} \in \mathrm{share}$ **do**
  if $\mathrm{ActualKey}[\mathrm{u}, \mathsf{e}]$ **then** $\mathrm{keys}(\mathsf{e}) \leftarrow \mathsf{Keys}[\mathsf{e}]$
return keys

Oracle Grant

Input: $(\mathrm{u}, \mathrm{share}, \mathrm{leak}) \in [\mathrm{n}] \times [\mathrm{n}] \times \mathcal{P}(\mathbb{N}) \times \{0,1\}$
req $\mathcal{G}(\mathsf{KnownSeed}[\mathrm{u},\cdot], \mathsf{KnownKey}[\mathrm{u},\cdot], \mathrm{share})$
$\mathrm{actual} \leftarrow \{\mathsf{e} \in \mathrm{share} \mid \mathrm{ActualKey}[\mathrm{u}, \mathsf{e}]\}$
req $\neg\mathrm{leak} \vee \mathrm{actual} \subseteq \mathit{Corr}$
try $(\mathrm{msg}; \bot) \leftarrow \langle \mathrm{U.Grant}(\mathsf{St}[\mathrm{u}], \mathrm{share}) \leftrightarrow \mathcal{A} \rangle$
if leak **then**
  $\mathrm{h} \leftarrow \mathrm{msg}$
else
  $\mathrm{h} \leftarrow\$ \{0,1\}^\kappa$ // handle for delivery
  $\mathrm{Msgs}[\mathrm{h}] \leftarrow (\mathrm{msg}, \mathrm{actual})$
return h

Oracle Accept

Input: $(\mathrm{u}', \mathrm{share}, \mathrm{h}) \in [\mathrm{n}] \times \mathcal{P}(\mathbb{N}) \times \{0,1\}^*$
$\mathrm{actual} = \bot$
if $\mathrm{Msgs}[\mathrm{h}] \neq \bot$ **then**
  $(\mathrm{msg}, \mathrm{actual}) \leftarrow \mathrm{Msgs}[\mathrm{h}]$ // Delivery
else
  $\mathrm{msg} \leftarrow \mathrm{h}$ // Injection
try $(\mathsf{St}[\mathrm{u}'], \mathrm{st_{up}}; \bot)$
  $\leftarrow \langle \mathrm{U.Accept}(\mathsf{St}[\mathrm{u}'], \mathrm{share}, \mathrm{msg}, \mathrm{upload}) \leftrightarrow \mathcal{A} \rangle$
try $\langle \mathrm{U.Upload}(\mathrm{st_{up}}) \leftrightarrow \mathcal{A} \rangle$
if $\mathrm{actual} \neq \bot$ **then**
  for $\mathsf{e} \in \mathrm{actual}$ **do**
    if $\mathsf{KnownKey}[\mathrm{u}', \mathsf{e}] = \texttt{false}$ **then**
      $\mathrm{ActualKey}[\mathrm{u}', \mathsf{e}] \leftarrow \texttt{true}$
for $\mathsf{e} \in \mathrm{share}$ **do**
  $\mathsf{KnownKey}[\mathrm{u}', \mathsf{e}] \leftarrow \texttt{true}$

$\texttt{sample-if-nec}(\mathrm{U}, \mathsf{e}, \mathrm{s})$

if $\mathrm{s} \neq \bot$ **then**
  $\mathrm{k} \leftarrow \mathrm{U.Key}(\mathsf{e}, \mathrm{s})$
  return (s, k)
else if $\mathsf{Secret}[\mathsf{e}] \neq \bot$ **then**
  $\mathsf{Secret}[\mathsf{e}] \leftarrow\$ \{0,1\}^\kappa$
  if $b = 0 \vee \mathsf{e} \in \mathit{Corr}$ **then** $\mathsf{Keys}[\mathsf{e}] \leftarrow \mathrm{U.Key}(\mathsf{e}, \mathrm{s})$
  else $\mathsf{Keys}[\mathsf{e}] \leftarrow\$ \{0,1\}^\kappa$
$\mathrm{ActualKey}[\mathrm{U}, \mathsf{e}] \leftarrow \texttt{true}$
return $(\mathsf{Secret}[\mathsf{e}], \mathsf{Keys}[\mathsf{e}])$

Fig. 2: The key-indistinguishability notion for standard-model CKS. We assume the adversary to not interleave calls of the adversarial oracles for the same user.

not to interleave calls of the oracles for the same user.) The game mostly just executes the protocol while keeping track of some additional state. For instance, $\mathsf{KnownSeed}[\mathrm{U}, \mathsf{e}]$ and $\mathsf{KnownKey}[\mathrm{U}, \mathsf{e}]$ keep track whether the user u knows the seed and key for epoch e, respectively. Furthermore, the game uses $\mathrm{ActualKey}[\mathrm{u}, \mathsf{e}]$ to keep track whether the key known by the user for an epoch is the one chosen by the game or one injected by the adversary. The seeds and keys chosen by the game are tracked using $\mathsf{Seed}[\mathsf{e}]$ and $\mathsf{Keys}[\mathsf{e}]$.

Observe that the function `sample-if-nec` samples one fresh seed for every epoch e and then uses that one consistently thorough the execution. Furthermore, depending on the bit b, it either derives the respective key or chooses an independent one. Those keys are then output as part of U.Append and U.Retrieve as challenges, whenever the respective user is known to use the proper seed. (If the adversary instead provides a seed for U.Append, by inputting $\mathrm{s} \neq \bot$, then `sample-if-nec` just returns keys that are consistent.)

Finally, note that in the Grant oracle the adversary gets to choose whether they receive the message produces by U.Grant or not, using the leak flag. This message is assumed to be transmitted over a secure channel to the recipient; therefore, leaking the message implies leaking the delegated keys. If the message is transmitted securely, then the adversary is given an opaque handle h to the message instead, which they later can use for delivery to another user u'. In the Accept oracle, the adversary can then either input a handle h or a granting message msg. In the former case, the game looks up the actual granting message msg, as well as for which of the granted keys correspond to actual keys chosen by the game (as opposed to keys injected by the adversary).

Note that the game formalizes *selective security* by having the adversary commit to the set Corr of all epochs for which the keys can be compromised before starting the interaction. Any type of corruption, whether leaking a user's private set or a grant message, is then predicated all of the keys which can be deduced by correctness being for corruptible epochs. Finally, observe that this formalizes forward secrecy and post-compromise security as corruptions are allowed whenever a user are not supposed to know the epoch's actual key.

3 Trapdoor Key Derivation

3.1 Defining TKDFs

In this section, we introduce the Trapdoor KDF (TKDF) primitive that will serve as the fundamental building block for our standard-model schemes. Recall from Section 1.1 that TKDF, in a nutshell,

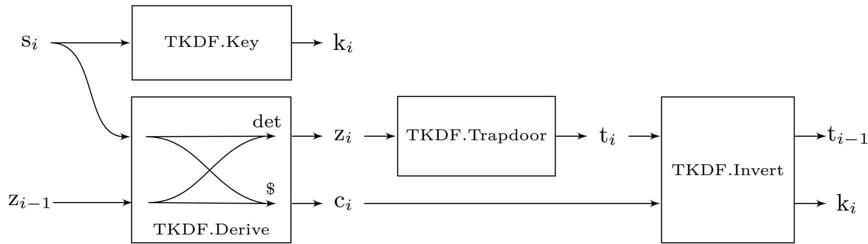


Fig. 3: A schematic representation of our symmetric TKDF scheme.

represents a kind of "invertible" KDF that generates a key k and updated state z' from a seed s and previous state z, such that from z a secret *trapdoor* t to invert the operation can be derived.

A schematic representation of a TKDF scheme is presented in Fig. 3. Observe that (to avoid circularity) we only require the inversion to recover the key and trapdoor rather than the seed and state. To be suitable for our CKS construction, we also require certain operations to be deterministic, as expressed as part of the following definition.

Definition 5. *A* Trapdoor Key Derivation Function (TKDF) *is a tuple* (TKDF. Key, TKDF. Derive, TKDF. Trapdoor, TKDF. Invert) *of PPT algorithms with an associated seed space $\mathcal{S}$, key space $\mathcal{K}$, state space $\mathcal{Z}$, trapdoor space $\mathcal{T}$, and ciphertext space $\mathcal{C}$.*

- *The deterministic key derivation algorithm* $\mathrm{k}_i \leftarrow \mathrm{TKDF.Key}(\mathrm{s}_i)$ *outputs the key* k_i *corresponding to a seed* s_i
- *The state derivation algorithm* $(\mathrm{z}_i, \mathrm{c}_i) \leftarrow \mathrm{TKDF.Derive}(\mathrm{s}_i, \mathrm{z}_{i-1})$ *takes a seed* s_i *and a state* z_{i-1} *as inputs, and outputs the next state* z_i *and a ciphertext* c_i*. The algorithm can be randomized, but* z_i *is a deterministic function of the inputs. Hence, only* c_i *may depend on the algorithm's randomness.*
- *The deterministic* $\mathrm{t}_i \leftarrow \mathrm{TKDF.Trapdoor}(\mathrm{z}_i)$ *algorithm outputs a trapdoor* t_i *based on the state.*
- *The deterministic inversion algorithm* $(\mathrm{k}_i, \mathrm{t}_{i-1}) \leftarrow \mathrm{TKDF.Invert}(\mathrm{c}_i, \mathrm{t}_i)$ *takes the ciphertext and trapdoor, and outputs the key and the previous trapdoor.*

We generally require the state space $\mathcal{Z}$ to be small (e.g., about as big as the key space) and in particular to consist only of elements of the same length.

Correctness requires that TKDF. Invert correctly inverts TKDF. Derive, as formalized by the following definition.

Definition 6 (Correctness). *We say that a TKDF is correct, if*

$$\Pr\left[(\mathrm{k}_i, \mathrm{t}_{i-1}) = \mathrm{TKDF.Invert}(\mathrm{c}_i,\ \mathrm{t}_i) \;\middle|\; \begin{array}{r} \mathrm{s}_i \leftarrow\!\$\ \mathcal{S},\ \ \mathrm{z}_{i-1} \leftarrow\!\$\ \mathcal{Z}, \\ \mathrm{t}_{i-1} \leftarrow \mathrm{TKDF.Trapdoor}(\mathrm{z}_{i-1}), \\ \mathrm{k}_i \leftarrow \mathrm{TKDF.Key}(\mathrm{s}_i), \\ (\mathrm{z}_i, \mathrm{c}_i) \leftarrow \mathrm{TKDF.Derive}(\mathrm{s}_i, \mathrm{z}_{i-1}), \\ \mathrm{t}_i \leftarrow \mathrm{TKDF.Trapdoor}(\mathrm{z}_i) \end{array}\right] = 1,$$

where the randomness is taken both over the sampling of z_{i-1} *and* s_i*, as well as over the coins of* TKDF. Derive*.*

For security, we require that the resulting key k_i is indistinguishable from an independent uniform random, for an attacker that does not know the seed s_i. Analogously, we require that the trapdoor t_i is indistinguishable from random for an attacker that does not know the secret state z_i. (This will be important for being able to iterate the TKDF.) Finally, we require the ciphertext c_i to be semantically secure and not reveal any information about either k_i or t_{i-1} to an adversary not knowing t_i. The precise security definitions are a bit subtle. We now first state the formal definition and then discuss some of the intricacies.

Definition 7 (TKDF Security). *A TKDF scheme is said to be secure if there exists a PPT algorithm* $\mathrm{TKDF.Sim}(\mathrm{t}_i, \mathrm{k}_i, \mathrm{t}_{i-1}) \to \mathrm{c}_i$ *which simulates ciphertexts such that the following three properties hold:*

1. ***Key randomness.*** *For any PPT $\mathcal{A}$, the advantage*

$$\mathsf{Adv}_{\mathsf{G}^{\mathrm{KeyRand}}_{\mathrm{TKDF}}}(\mathcal{A}) := \left|\Pr[\mathsf{G}^{\mathrm{KeyRand\text{-}0}}_{\mathrm{TKDF}}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathsf{G}^{\mathrm{KeyRand\text{-}1}}_{\mathrm{TKDF}}(\mathcal{A}) \Rightarrow 1]\right|$$

is negligible in the security parameter κ, for the real-or-ideal game from Fig. 4.

2. ***Trapdoor randomness.*** *For any PPT $\mathcal{A}$, the advantage*

$$\mathsf{Adv}_{\mathsf{G}^{\mathrm{TdRand}}_{\mathrm{TKDF}}}(\mathcal{A}) := \left|\Pr[\mathsf{G}^{\mathrm{TdRand\text{-}0}}_{\mathrm{TKDF}}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathsf{G}^{\mathrm{TdRand\text{-}1}}_{\mathrm{TKDF}}(\mathcal{A}) \Rightarrow 1]\right|$$

is negligible in the security parameter κ, for the real-or-ideal game from Fig. 5.

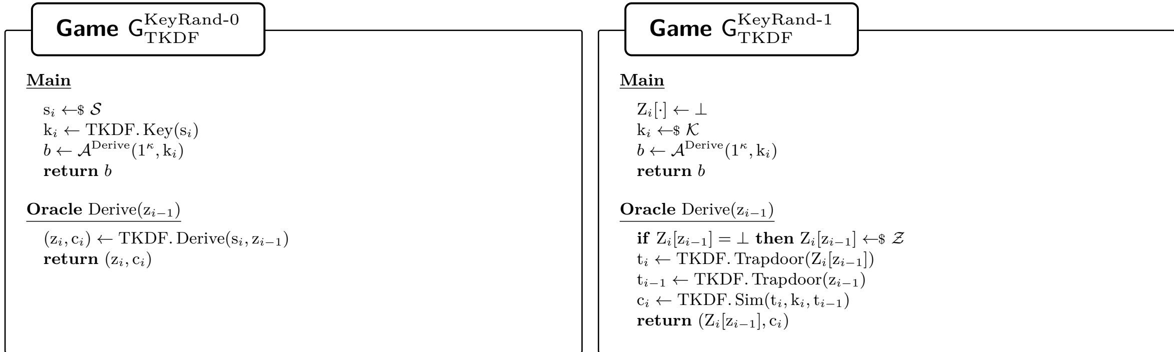


Fig. 4: The games formalizing key-randomness of a TKDF scheme.

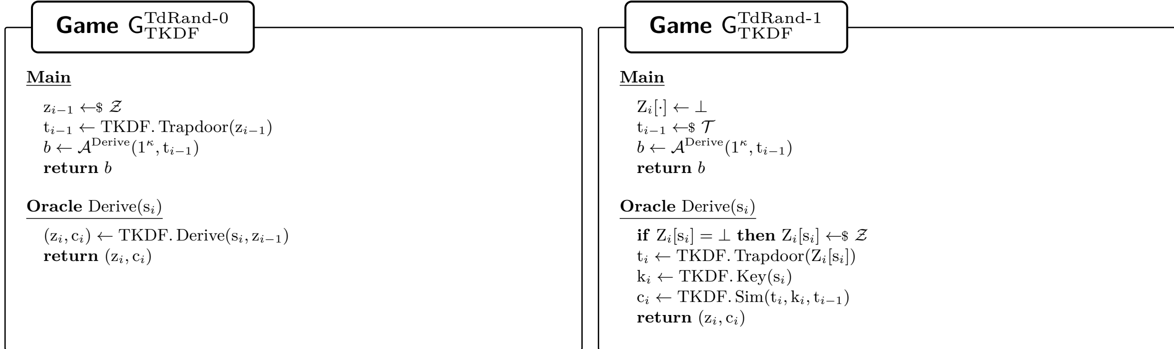


Fig. 5: The games formalizing trapdoor-randomness of a TKDF scheme.

3. ***Semantic security.*** *For any keys* k_i^0 *and* k_i^1*, and any trapdoors* t_{i-1}^0 *and* t_{i-1}^1*, the following distributions are computationally indistinguishable:*

$$\left\{\left(k_i^0, k_i^1, t_{i-1}^0, t_{i-1}^1, \mathrm{TKDF.Sim}(t_i, k_i^0, t_{i-1}^0)\right) : t_i \leftarrow\!\!\$\ \mathcal{T}\right\} \approx_c \left\{\left(k_i^0, k_i^1, t_{i-1}^0, t_{i-1}^1, \mathrm{TKDF.Sim}(t_i, k_i^1, t_{i-1}^1)\right) : t_i \leftarrow\!\!\$\ \mathcal{T}\right\}.$$

We denote with $\mathsf{Adv}_{\mathsf{G}_{\mathrm{TKDF}}^{\mathrm{IND\text{-}CPA\text{-}OT}}}(\mathcal{A})$ *the maximum respective advantage for an adversary* $\mathcal{A}$*, over any challenge.*

We say that the TKDF scheme is one-time *secure if no PPT adversary has non-negligible advantage when restricted to a single* Derive *query in the key-randomness and trapdoor-randomness games.*

Let us consider the key-randomness property. Intuitively, this property requires that k_i is indistinguishable from random when not knowing the seed s_i. However, the property further has to account for leakage from the ciphertext and, especially, the state z_i (which can be used for subsequent evaluations). Therefore, the property demands that k_i and z_i are indistinguishable from independently sampled uniform random values. In any actual scheme, however, those values are related via the ciphertext c_i by correctness: Any attacker can derive t_i from z_i and use this to decrypt c_i, resulting in (t_{i-1}, k_i). Therefore, $\mathsf{G}_{\mathrm{TKDF}}^{\mathrm{KeyRand\text{-}1}}$ allows the ciphertext to be generated consistently using a simulator TKDF. Sim. Crucially, the simulator ensures that t_i from z_i are only related indirectly via the trapdoor t_i, i.e., that the above check is essentially the only thing an attacker can do to distinguish k_i and z_i from independent uniform random values. Finally, note that the attacker gets multiple TKDF. Derive queries for their prior state z_{i-1} of choice. This will turn out to be vital for active security where several users knowing the same seed s_i are tricked to evaluate the TKDF with different prior states. Indeed, for passive security one-time TKDF security suffices.

Trapdoor randomness is then defined analogously to key randomness. For instance, the attacker gets to do multiple TKDF. Derive to anticipate attacks where to parties with the same secret state z_i are tricked into using different seeds s_i chosen by the adversary.

Finally, consider semantic security. This is essentially one-time IND-CPA security for the ciphertext. Note that as all three security properties use the same simulator, the former ones already imply that the simulated ciphertext is indistinguishable from a real one. Phrasing semantic security in terms of the simulator will turn out to make the definition a bit easier to use in hybrid arguments where either key randomness or trapdoor randomness is applied first.

3.2 Symmetric TKDF

For more CKS schemes with non-trivial delegation and erasure, we need a TKDF that treats its two inputs more interchangeably than the basic TKDF notion introduced above. That is, a TKDF does not strictly distinguish between the concepts of seeds and states and allows them to be used somewhat interchangeably. In particular, we want (1) a TKDF state z can be used as a seed for a subsequent TKDF call, and (2) that trapdoors are generated analogously to derived keys.

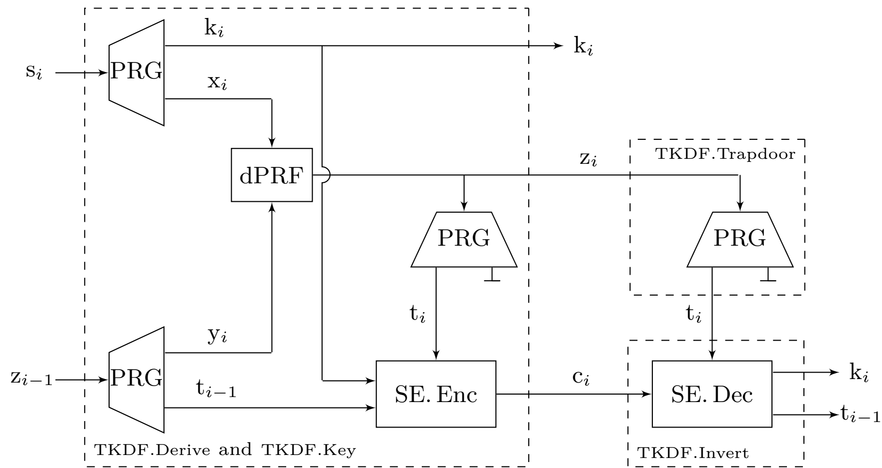
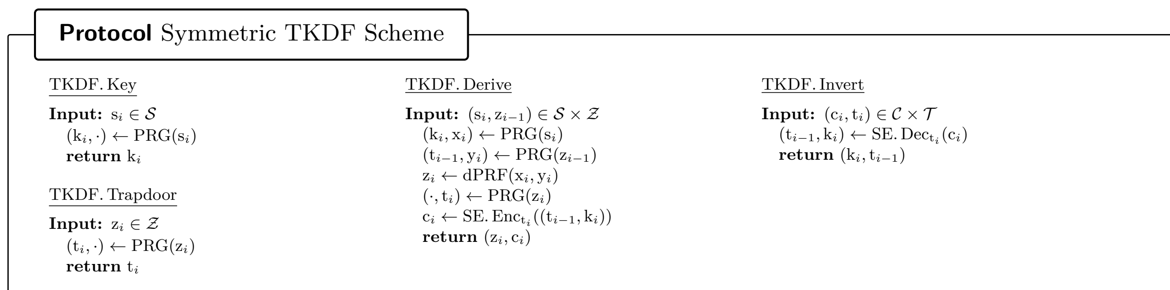


Fig. 6: A schematic representation of the Symmetric TKDF scheme from Fig. 7. The left box shows both TKDF. Key (the upper output only depending on s_i) and TKDF. Derive (the middle and lower outputs).


Protocol Symmetric TKDF Scheme

TKDF. Key

Input: $s_i \in \mathcal{S}$
$(k_i, \cdot) \leftarrow \mathrm{PRG}(s_i)$
return k_i

TKDF. Trapdoor

Input: $z_i \in \mathcal{Z}$
$(t_i, \cdot) \leftarrow \mathrm{PRG}(z_i)$
return t_i

TKDF. Derive

Input: $(s_i, z_{i-1}) \in \mathcal{S} \times \mathcal{Z}$
$(k_i, x_i) \leftarrow \mathrm{PRG}(s_i)$
$(t_{i-1}, y_i) \leftarrow \mathrm{PRG}(z_{i-1})$
$z_i \leftarrow \mathrm{dPRF}(x_i, y_i)$
$(\cdot, t_i) \leftarrow \mathrm{PRG}(z_i)$
$c_i \leftarrow \mathrm{SE.Enc}_{t_i}((t_{i-1}, k_i))$
return (z_i, c_i)

TKDF. Invert

Input: $(c_i, t_i) \in \mathcal{C} \times \mathcal{T}$
$(t_{i-1}, k_i) \leftarrow \mathrm{SE.Dec}_{t_i}(c_i)$
return (k_i, t_{i-1})

Fig. 7: A simple construction based on a PRG, a dual-PRF, and symmetric encryption. Note that TKDF. Key and TKDF. Trapdoor are the same, as required for a symmetric TKDF.

We call such a TKDF a symmetric TKDF. (We remark that unlike the notion of a symmetric PRF [4] we do not necessarily require that such a TKDF treats its argument symmetrically, i.e., TKDF. Derive(s, z) = TKDF. Derive(z, s), but simply that the two arguments play the same general role.) We formalize the structural requirement in the following definition.

Definition 8. *A* symmetric TKDF *is a TKDF scheme with the following structural properties:*

– *The seed space is equal to the state space, i.e., $\mathcal{S} = \mathcal{Z}$, and the key space is equal to the trapdoor space, i.e., $\mathcal{K} = \mathcal{T}$.*
– *The key derivation is equivalent to the trapdoor derivation, i.e.,* TKDF. Key = TKDF. Trapdoor.

The security of a symmetric TKDF is the same as the security of a regular TKDF. Note that key randomness and trapdoor randomness coincide iff TKDF. Derive treats its argument symmetrically.

3.3 A Standard-Model Symmetric-TKDF Construction

We now present a simple construction of a symmetric TKDF. The construction is based on a length-doubling PRG, a dual-PRF [2,4], and a one-time secure symmetric encryption scheme SE. Recall that a dual-PRF is a deterministic algorithm $\mathrm{dPRF} : \{0,1\}^\kappa \times \{0,1\}^\kappa \to \{0,1\}^\kappa$ that behaves like a PRF in both arguments, i.e., such that for a uniform random key k, both $\mathrm{dPRF}(k, \cdot)$ and $\mathrm{dPRF}(\cdot, k)$ are PRFs. The scheme first uses the PRG to expand the seed s_i into the key k_i and an auxiliary value x_i, and to expand the previous z_{i-1} into y_i and the previous trapdoor t_{i-1}. The values x_i and y_i are then combined using the dPRF to obtain the next state z_i. From this state, we moreover derive the current trapdoors t_i as the second part of the output yield from expanding z_i (analogously as for t_{i-1}). t_i then serves as encryption key to encrypt (t_{i-1}, k_i). The trapdoor algorithm simply recomputes t_i from z_i and, finally, the inversion algorithm decrypts the ciphertext to obtain the trapdoor and key.[5] A schematic depiction of the scheme is presented in Fig. 6, whereas for completeness formal definition is given in Fig. 7.

Note that we assumed here that $\mathcal{Z} = \mathcal{S} = \{0,1\}^\kappa$. Using a length-doubling PRG, we can thus observe that $\mathcal{K} = \mathcal{T} = \{0,1\}^\kappa$ as well, implying that our construction satisfies the structural property of a symmetric TKDF. Correctness then immediately follows by the correctness of the symmetric encryption scheme SE. Security is established by the following theorem.

[5] We remark that if a party consecutively computes $z_i \leftarrow \mathrm{TKDF.Derive}(s_i, z_{i-1})$, and $z_{i+1} \leftarrow \mathrm{TKDF.Derive}(s_{i+1}, z_i)$, then an actual implementation would not need to expand z_i in both operations separately. Thus, the number of PRG iterations per epoch is actually two instead of three.

Theorem 1. *Assume* SE *is a one-time IND-CPA secure symmetric encryption scheme,* PRG *is a secure length-doubling pseudo-random generator, and* dPRF *is a secure dual-PRF. Then, the TKDF from Fig. 7 scheme is secure. More formally, for each of the three properties and every PPT adversary* $\mathcal{A}$*, there exist attackers* $\mathcal{A}_{\mathrm{PRG}}$ *against the PRG security game* $\mathsf{G}^{\mathrm{IND}}_{\mathrm{PRG}}$*, and* $\mathcal{A}_{\mathrm{SE}}$ *against the IND-CPA game* $\mathsf{G}^{\mathrm{IND\text{-}CPA}}_{\mathrm{SE}}$*, that have roughly the same running time, such that*

$$\mathsf{Adv}_{\mathsf{G}^{\mathrm{KeyRand}}_{\mathrm{TKDF}}}(\mathcal{A}) \le \mathsf{Adv}_{\mathsf{G}^{\mathrm{IND}}_{\mathrm{PRG}}}(\mathcal{A}_{\mathrm{PRG}}) + \mathsf{Adv}_{\mathsf{G}^{\mathrm{IND}}_{\mathrm{dPRF}}}(\mathcal{A}_{\mathrm{dPRF}}) \tag{1}$$

$$\mathsf{Adv}_{\mathsf{G}^{\mathrm{TdRand}}_{\mathrm{TKDF}}}(\mathcal{A}) \le \mathsf{Adv}_{\mathsf{G}^{\mathrm{IND}}_{\mathrm{PRG}}}(\mathcal{A}_{\mathrm{PRG}}) + \mathsf{Adv}_{\mathsf{G}^{\mathrm{IND}}_{\mathrm{dPRF}}}(\mathcal{A}_{\mathrm{dPRF}}) \tag{2}$$

$$\mathsf{Adv}_{\mathsf{G}^{\mathrm{IND\text{-}CPA\text{-}OT}}_{\mathrm{TKDF}}}(\mathcal{A}) \le \mathsf{Adv}_{\mathsf{G}^{\mathrm{IND\text{-}CPA\text{-}OT}}_{\mathrm{SE}}}(\mathcal{A}_{\mathrm{SE}}). \tag{3}$$

Proof. We use the following simple simulator that just mimics the encryption performed by the scheme

$$\mathrm{TKDF.Sim}(\mathrm{t}_i, \mathrm{k}^0_i, \mathrm{t}^0_{i-1}) := \mathrm{SE.Enc}_{\mathrm{t}_i}((\mathrm{t}_{i-1}, \mathrm{k}_i)).$$

First, consider key randomness. Observe that, in the TKDF. Derive computation of $\mathsf{G}^{\mathrm{KeyRand\text{-}0}}_{\mathrm{TKDF}}$, by PRG security k_i and x_i are indistinguishable from independent and uniformly random sampled values as s_i is sampled uniformly at random and not otherwise used. Now, we can apply dual-PRF security to conclude that in the Derive oracle the $\mathrm{dPRF}(\mathrm{x}_i, \mathrm{z}_{i-1})$ evaluation furthermore behaves like a uniform random function in the second argument. Therefore, we can instead replace z_i with the output of a URF, as in $\mathsf{G}^{\mathrm{KeyRand\text{-}1}}_{\mathrm{TKDF}}$. The indistinguishability now follows by observing that TKDF. Derive just computes t_i and t_{i-1} the same way as TKDF. Trapdoor in $\mathsf{G}^{\mathrm{KeyRand\text{-}1}}_{\mathrm{TKDF}}$, followed by the same encryption that TKDF. Sim performs.

The trapdoor randomness follows analogously, using that $\mathrm{dPRF}(\cdot, \mathrm{z}_{i-1})$ behaves like a uniform random function for z_{i-1} chosen uniformly at random. Finally, consider the semantic security. Given the definition of our simulator, this follows directly from the one-time IND-CPA security of SE. Enc. □

Corollary 1. *Trapdoor KDFs exist if and only if dual-PRFs exist.*

Proof. Dual-PRFs imply the existence of one-way functions and, thus, PRG and (one-time secure) symmetric encryption. Therefore, the first direction follows from Theorem 1. Moreover, observe that key-randomness and trapdoor-randomness games jointly imply dual-PRF security with respect to the first output z_i of TKDF. Derive.

Variants. We now consider some variants of the scheme. First, observe that for one-time TKDF security, we can replace the dual-PRF with a simple XOR operation of x_i and y_i. The proof follows analogously, observing that for a single evaluation $\mathrm{x}_i \oplus \mathrm{y}_i$ behaves indistinguishable from the dual-PRF. While dPRFs are known to be constructible from standard assumptions [4] it is an open problem whether than can be constructed from one-way functions only.

Theorem 2. *When replacing* $\mathrm{z}_i \leftarrow \mathrm{dPRF}(\mathrm{x}_i, \mathrm{y}_i)$ *with* $\mathrm{z}_i \leftarrow \mathrm{x}_i \oplus \mathrm{y}_i$ *in the scheme from Fig. 7, then the modified scheme is one-time TKDF secure, assuming* SE *is a one-time IND-CPA secure symmetric encryption scheme and* PRG *is a secure length-doubling pseudo-random generator.*

Corollary 2. *The existence of one-way functions implies the existence of one-time secure TKDF schemes.*

Second, we observe that the usage of the PRG is just one special case of a key derivation mechanism, expanding a seed s_i or state z_i into two independent secrets. Therefore, if we have a legacy application that already prescribes a key-derivation step, and that allows us to derive one more unrelated secret, then our scheme can be made compatible with the legacy application. For instance, the MLS group messaging protocol already involves a key derivation function (KDF) based on HKDF [14,5]. Our TKDF scheme, could therefore derive k_i according to that key derivation and x_i using the same key derivation but on a different context (and analogously for expanding z_i). As long as the legacy application does not use that context itself, the composed scheme is secure. The proof of the following theorem follows analogous to Theorem 1.

Theorem 3. *For any secure key derivation function (KDF) we can replace the usage of* PRG *in the scheme from Fig. 7 with the* KDF *evaluated twice on two distinct contexts. The resulting scheme is a secure TKDF, assuming the KDF, the dual-PRF are secure and* SE *is a one-time IND-CPA secure symmetric encryption scheme.*

4 Iterative CKS

Recall from Section 1.1 that the all-or-nothing scheme by Dodis et al. was built around iteratively applying a "derivation" that aggregates a secret state and a seed into a new secret state (and a ciphertext) as depicted in Fig. 1. The second scheme by Dodis et al. — which allows for efficient delegation and erasure of arbitrary continuous intervals of secrets — follows a similar template. Indeed, the scheme simply arranges the epochs as leaves in a binary tree, where each node aggregates its two children. (We refer to [7] for details on the scheme.)

In this section, we abstract CKS schemes built around this template. We call such a scheme an *iterative* CKS scheme, for which we define the respective notion. The corresponding (security) definitions for this special case turn out to be significantly simpler than the (fully general) CKS notion as introduced in [7]. We then present a unified protocol for the iterative CKS template, based on a TKDF, and prove its security. This essentially allows us to reduce choosing the right trade-off between functionality (in terms of delegation and fine-grained erasure) and efficiency of CKS schemes to a graph theoretic problem.

4.1 Syntax

In this section, we formally introduce the simplified iterative CKS notion. On a high level, the idea is that such a scheme repeatedly aggregates secrets — which could either be seeds or secret states themselves — into a new secret and a ciphertext. Intuitively, the resulting secret should allow to reverse the aggregation. Therefore, recursively, the state when combined with the appropriate ciphertexts should allow recovering any secret that went into the aggregation.

Note that the aggregation essentially forms a directed graph with seeds as sources and each non-source having indegree two. To avoid circularity, we will restrict ourselves to directed acyclic graphs (DAG). The following definition assigns each node the set of epochs that have a path from their source to the node, i.e., the set of seeds they aggregate over.

Definition 9. *We say* $\mathfrak{S} \subseteq \mathcal{P}(2^{\mathbb{N}})$ *is a* set family *for an interactive CKS if:*

1. $\varnothing \notin \mathfrak{S}$
2. $\{\mathsf{e}\} \in \mathfrak{S}$*, for all* $\mathsf{e} \in \mathbb{N}$
3. *For each set* $\mathcal{S} \in \mathfrak{S}$ *with* $|\mathcal{S}| > 1$*, there exists a unique decomposition* $\mathcal{S}_1, \mathcal{S}_2 \in \mathfrak{S}$ *such that* $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$*.*

Furthermore, let $\mathcal{D}ag_{\mathfrak{S}}$ *denote the respective DAG over the set* $\mathfrak{S}$ *where an edge* $(\mathcal{S}_i, \mathcal{S}_j)$ *is present iff there exists* $\mathcal{S}' \in \mathfrak{S}$ *such that* $\mathcal{S}_j = \mathcal{S}_i \cup \mathcal{S}'$*. By property 3, each internal node of* $\mathcal{D}ag_{\mathfrak{S}}$ *has in-degree* 2*.*

We now define iterative CKS for a set family $\mathfrak{S}$. Recursively aggregating seeds according to edges in $\mathcal{D}ag_{\mathfrak{S}}$, such a scheme allows a party knowing all seeds $\{\mathrm{s}_\mathsf{e} \mid \mathsf{e} \in \mathcal{S}\}$ to create a compact *seeds state* $SS_\mathcal{S}$ and a compact *keys state* $KS_\mathcal{S}$, as well as ciphertext $C_\mathcal{S}$. For security, the ciphertexts should not reveal any information about the keys derived from the seeds. For correctness, on the other hand, the keys state $KS_\mathcal{S}$ and ciphertext $C_\mathcal{S}$ should be sufficient to recover the *keys* by "undoing" the aggregation and, ultimately, recover individual keys k_e.

Definition 10. *An* Iterative Compact Key Storage (I-CKS) *scheme* CKS *consists of the following PPT algorithms:*

- $\mathrm{GenPub}(1^\kappa) \to \mathrm{pub}$ *generates public parameters for the scheme.*
- $\mathrm{DeriveKey}(\mathrm{pub}, \mathrm{s}_\mathsf{e}, \mathsf{e}) \to \mathrm{k}_\mathsf{e}$ *returns the key corresponding to a seed for epoch* e*. This algorithm is assumed to be deterministic.*
- $\mathrm{Init}(\mathrm{pub}, \mathsf{e}, \mathrm{s}_\mathsf{e}) \to (SS_{\{\mathsf{e}\}}, C_{\{\mathsf{e}\}})$ *initializes a secret seeds state for* $\mathcal{S} = \{\mathsf{e}\}$*. Additionally, output an (optional) ciphertext.*
- $\mathrm{Compact}(\mathcal{S}_1, SS_{\mathcal{S}_1}, \mathcal{S}_2, SS_{\mathcal{S}_2}) \to (SS_{\mathcal{S}_1 \cup \mathcal{S}_2}, C_{\mathcal{S}_1 \cup \mathcal{S}_2})$ *takes two seed states and compacts them into a joint one and a ciphertext to be stored. This assumes that* $(\mathcal{S}_1 \cup \mathcal{S}_2) \in \mathfrak{S}$*.*
- $\mathrm{DeriveKS}(\mathcal{S}, SS_\mathcal{S}) \to KS_\mathcal{S}$ *computes the keys state corresponding to a seeds state.*
- $\mathrm{Expand}(\mathcal{S}, KS_\mathcal{S}, C_\mathcal{S}, \mathcal{S}_1, \mathcal{S}_2) \to (KS_{\mathcal{S}_1}, KS_{\mathcal{S}_2})$ *obtains the keys state for a subintervals* $\mathcal{S}_1 \subset \mathcal{S}$ *and* $\mathcal{S}_2 \subset \mathcal{S}$ *based on the keys state for the joint interval* $\mathcal{S} = \mathcal{S}_1 \cup \mathcal{S}_2$ *and the respective ciphertext.*
- $\mathrm{Recover}(\mathsf{e}, KS_{\{\mathsf{e}\}}, C_{\{\mathsf{e}\}}) \to \mathrm{k}_\mathsf{e}$ *recovers the key* k_e *for an epoch* e*.*

Correctness. We now formalize the correctness of our notion. Simply put, we require the following two properties:

1. Recover "undoes" Init. Consider an epoch $\mathsf{e} \in \mathbb{N}$. Then,
   - $(SS_{\{\mathsf{e}\}}, C_{\{\mathsf{e}\}}) \leftarrow \mathrm{Init}(\mathrm{pub}, \mathsf{e}, \mathrm{s}_\mathsf{e})$
   - $KS_{\{\mathsf{e}\}} \leftarrow \mathrm{DeriveKS}(\{\mathsf{e}\}, SS_{\{\mathsf{e}\}})$
   - $\mathrm{k}'_\mathsf{e} \leftarrow \mathrm{Recover}(\mathsf{e}, KS_{\{\mathsf{e}\}}, C_{\{\mathsf{e}\}})$

   outputs the correct key, i.e., $\mathrm{k}'_\mathsf{e} = \mathrm{DeriveKey}(\mathrm{pub}, \mathrm{s}_\mathsf{e}, \mathsf{e})$.

2. Expand "undoes" Compact. Consider an epoch sets $\mathcal{S}_1, \mathcal{S}_2 \in \mathfrak{S}$ such that their union is in $\mathfrak{S}$. Then,
   - $(SS_{\mathcal{S}'}, C_{\mathcal{S}'}) \leftarrow \mathrm{Compact}(\mathcal{S}_1, SS_{\mathcal{S}_1}, \mathcal{S}_2, SS_{\mathcal{S}_2})$
   - $KS_{\mathcal{S}'} \leftarrow \mathrm{DeriveKS}(\mathcal{S}', SS_{\mathcal{S}'})$
   - $(KS'_{\mathcal{S}_1}, KS'_{\mathcal{S}_2}) \leftarrow \mathrm{Expand}(\mathcal{S}', KS_{\mathcal{S}'}, C_{\mathcal{S}'}, \mathcal{S}_1, \mathcal{S}_2)$

   produces keys states such that $KS'_{\mathcal{S}_1}$ is interchangeable with $KS_{\mathcal{S}_1}$ obtained via $\mathrm{DeriveKS}(\mathcal{S}_1, SS_{\mathcal{S}_1})$, and $KS'_{\mathcal{S}_2}$ is interchangeable with $KS_{\mathcal{S}_2}$, respectively.

A formal version of correctness is presented in Fig. 8. Note that the game treats any seeds state, keys state, and ciphertext for the same epoch set $\mathcal{S} \in \mathfrak{S}$ interchangeably. In a deterministic scheme this is trivially achieved by each being unique — we however only formally require DeriveKey to be deterministic.

4.2 Security

For security, we consider two games: pseudorandomness and integrity. The pseudorandomness game is depicted in Fig. 9. The game allows the adversary to create an arbitrary number of seed states for a single epoch, using Init, and then to gradually accumulate seeds using Compact, returning the ciphertext to the adversary. For Init, the adversary obtains either the real key k_e, if $b = 0$, or a random one, if $b = 1$. Note that the adversary can also inject their own seed by inputting $\mathrm{s}_\mathsf{e} \neq \perp$, in which case the real key is used. Finally, note that, for simplicity, the game formalizes selective security, with the adversary having to commit to the set of corruptions $\mathcal{C}orr$ ahead of time. However, observe that the structure of our game (and I-CKS in general) is essentially one of a pebbling game on the graph $\mathcal{D}ag_{\mathfrak{S}}$. Therefore, the framework on adaptive security by Jafargholi et al. [10] and Kamath et al. [13] should allow us to get (quasi-polynomial) adaptive security for specific graphs $\mathcal{D}ag_{\mathfrak{S}}$.

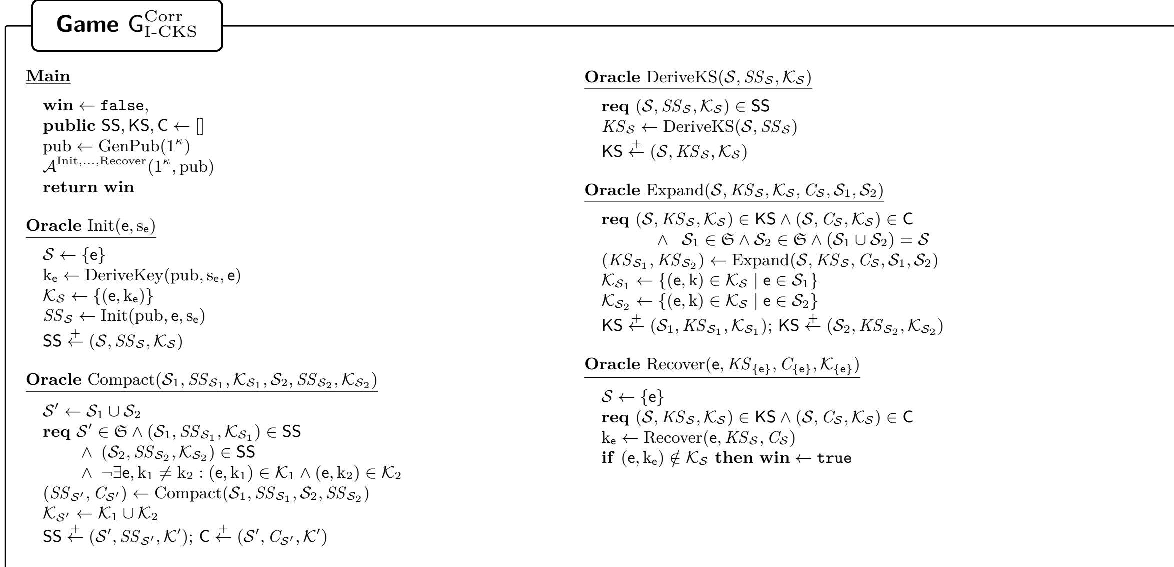


Fig. 8: The Iterative CKS correctness game.

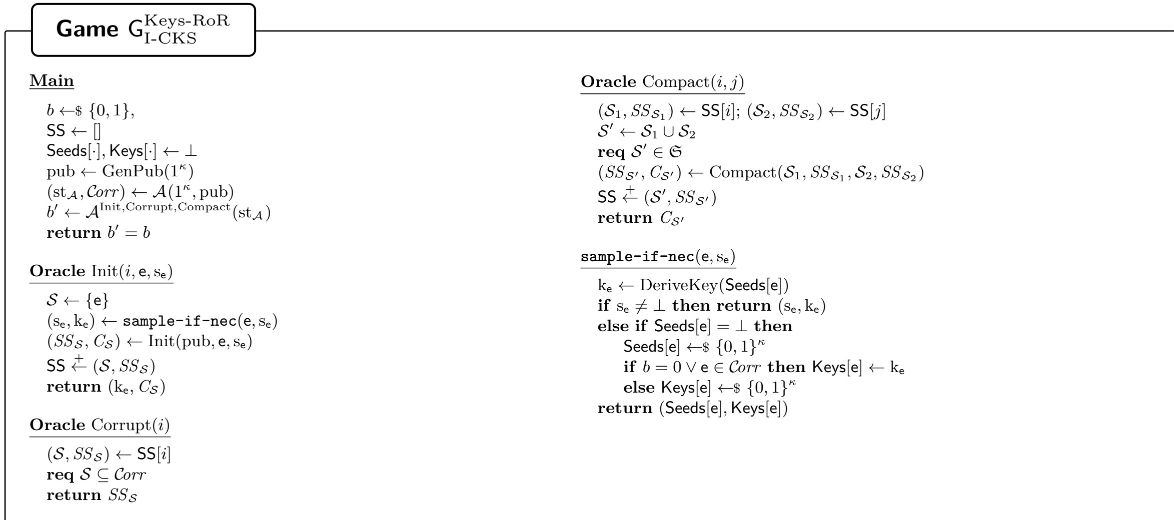


Fig. 9: The real-or-random security game for an Iterative CKS scheme.

Definition 11. *An I-CKS scheme is secure, if for any set family $\mathcal{S}$, the following advantage*

$$\mathsf{Adv}_{\mathsf{G}^{\text{Keys-RoR}}_{\text{I-CKS}}}(\mathcal{A}) := \Pr[\mathsf{G}^{\text{Keys-RoR}}_{\text{I-CKS}}(\mathcal{A}) \Rightarrow 1]$$

is negligible in κ for any PPT adversary $\mathcal{A}$. We say that the scheme is passively secure if the advantage is negligible for any $\mathcal{A}$ who is restricted to only pass $\mathrm{s}_{\mathsf{e}} = \bot$ to the Init *oracle.*

The integrity game, depicted in Fig. 10, is similar to the correctness game in structure. It, however, no longer restricts the adversary to submit honestly generated ciphertexts and, in turn, allows the algorithms to fail. Integrity then requires that Recover either fails or outputs the same key k_{e} that was initially aggregated over. In addition, the game can be won if Compact succeeds on seed states that contain conflicting information for the same epoch. Observe that the game formalizes a strong variant of integrity where all states are presumed to be public (analogous to the general CKS integrity game).

Definition 12. *An I-CKS scheme is said to satisfy integrity, if for any set family $\mathcal{S}$, the following advantage is negligible in κ for any PPT adversary $\mathcal{A}$:*

$$\mathsf{Adv}_{\mathsf{G}^{\text{Integrity}}_{\text{I-CKS}}}(\mathcal{A}) := \Pr[\mathsf{G}^{\text{Integrity}}_{\text{I-CKS}}(\mathcal{A}) \Rightarrow 1].$$

4.3 Constructing I-CKS from TKDF

We now build a generic iterative CKS scheme based on a (symmetric) TKDF. Recall from Section 1.1 that the goal of a TKDF was to provide a standard-model abstraction for the "derive" and "invert" boxes used in the schemes of [7]. See, for example, Fig. 1 for a high-level schematic of the ROM-CKS all-or-nothing scheme — and compare it with the intended analogous for the standard model presented in Fig. 11.

The scheme for a set family $\mathfrak{S}$ is then fairly straightforward. In a nutshell, TKDF states roughly correspond to seed states, and TKDF trapdoors to key states. For each internal node of $\mathcal{D}ag_{\mathfrak{S}}$ we use TKDF. Derive to compact the seed states of its two child nodes as part of Compact. Conversely, for Expand we use TKDF. Invert to obtain the key states of the node's children. We now discuss the scheme in a bit more detail; see Fig. 12 for a pseudocode description.

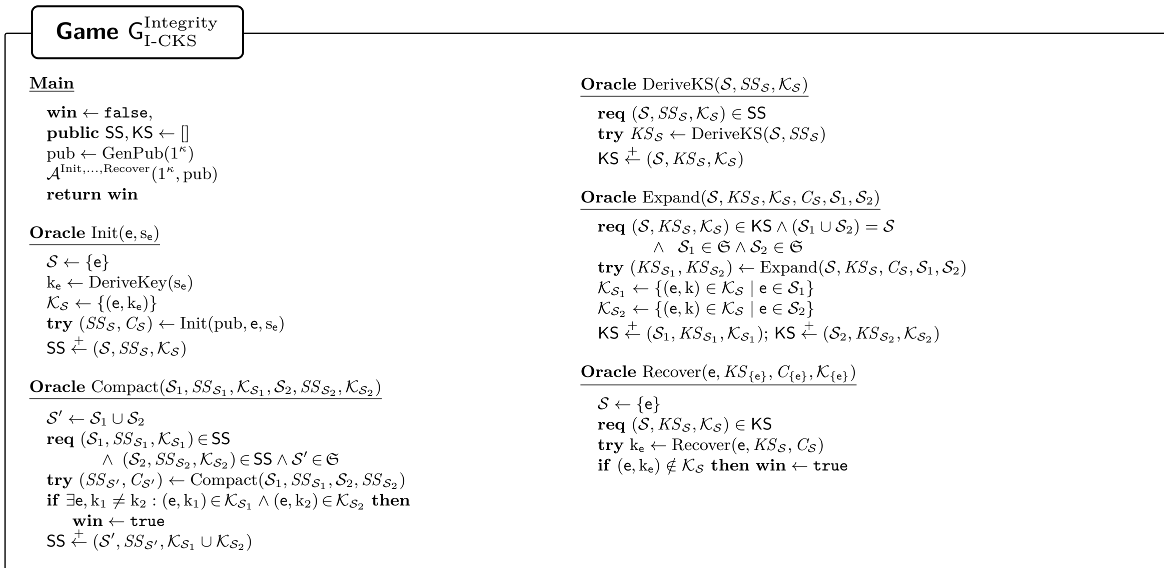


Fig. 10: The integrity game for an Iterative CKS scheme.

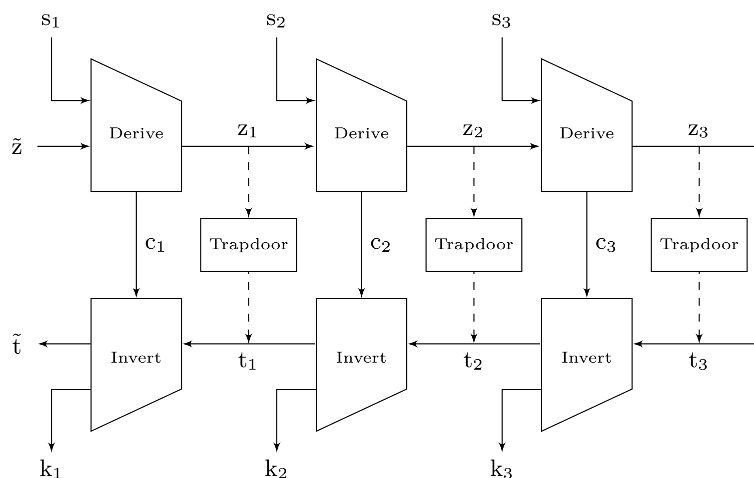


Fig. 11: The all-or-nothing scheme from Fig. 1 adapted to the standard-model CKS setting, using a TKDF instead of convergent encryption. $\tilde{z}$ is some fixed TKDF state used to initialize the iteration. Note that the tags T_i from Fig. 1 are omitted here (which are not to be confused with the trapdoors t_i).

Seed states and key states. For now, let us describe a variant of the scheme without integrity. Each seed state $SS_\mathcal{S}$ is then of one of two forms: (a) a regular state $SS_\mathcal{S} = z_\mathcal{S}$ storing a TKDF state, or (b) an immediate state $SS_\mathcal{S} = s_e$, in case $\mathcal{S} = \{e\}$. (For clarity, we further mark the state with the constant '`seeds`'.) Analogously, each keys state $KS_\mathcal{S}$ either (a) is a TKDF trapdoor $KS_\mathcal{S} = t_\mathcal{S}$, if $|\mathcal{S}| > 1$, or (b) a key $KS_e = k_e$ if $\mathcal{S} = \{e\}$.

Init and Recover. Both Init and Recover are, in principle, extremely simple. The former just outputs $SS_{\{e\}} = s_e$ as seed state, with no ciphertext necessary. The latter takes $KS_{\{e\}} = k_e$ and outputs k_e.

Some complications arise from supporting regular TKDF, for instance for the all-or-nothing scheme depicted in Fig. 11. Observe that here $SS_{\{1\}}$ and $SS_{\{2\}}$ have to behave slightly differently, as the former seed s_1 has the extra derivation with the initial constant state $\tilde{z}$. (Looking slightly ahead, the TKDF. Derive mixing in s_2 will be performed as part of Compact of $SS_{\{1\}}$ and $SS_{\{2\}}$.) To solve this issue, the formal protocol from Fig. 12 solves this issue by introducing a "base set" $\mathcal{B} \subseteq \mathbb{N}$ of epochs, for which such an extra derivation should be performed. Both $\mathcal{B}$ and $\tilde{z}$ are then considered protocol parameters.

Compact, DeriveKS, and Expand. As mentioned, Compact corresponds directly to TKDF. Derive and Expand to TKDF. Invert. Similarly, DeriveKS corresponds directly to TKDF. Trapdoor, deriving the TKDF trapdoor (= keys state) from the TKDF state (= seeds state). One more subtlety arises, however. We want different users to compact two states in the same order, i.e., they should agree on which of the states $SS_{\mathcal{S}_1}$ or $SS_{\mathcal{S}_2}$ is used as a first and which as a second argument. Otherwise, various parties who may learn the same set of seeds in different orders may still create incompatible outsourcings, undermining the compactness of the server state. In particular, the order must be well defined when using Expand to recover prior trapdoors. We solve this by introducing as a protocol parameter an order $\prec$ on any two sets $\mathcal{S}_1$ and $\mathcal{S}_2$ which can be combined. (Note that this is not required to be a proper order relation among all $\mathfrak{S}$, something like the lexicographic order on the descriptions of $\mathcal{S}_1$ and $\mathcal{S}_2$ would suffice.)

Protocol I-CKS Scheme

Parameters:
- set family $\mathfrak{S}$;
- ordering of sets $\mathcal{S}_1 \prec \mathcal{S}_2$, for all $\mathcal{S}_1, \mathcal{S}_2$ such that $\mathcal{S}_1 \cup \mathcal{S}_2 \in \mathfrak{S}$;
- set of epochs $\mathcal{B} \subseteq \mathbb{N}$ for which an extra derivation step is applied;
- a constant TKDF state $\tilde{z} \in \mathcal{Z}$ to be used for epochs $\mathsf{e} \in \mathcal{B}$.

```
GenPub(1^κ)
    pub ←$ {0,1}^κ
    return pub

DeriveKey(pub, s_e, e)
    k_e ← TKDF.Key(s_e)
    return k_e

Init(pub, e, s_e)
    h ← H_pub(e)
    if e ∈ B then
        (z, c) ← TKDF.Derive(s_e, z̃)
        SS ← ('seeds', z, h)
        C ← (h, ⊥, c)
        return (SS, C)
    else
        h ← H_pub(e)
        SS ← ('seeds', s_e, h)
        return (SS, ⊥)

Compact(S_1, SS_{S_1}, S_2, SS_{S_2})
    req S_1, S_2 ∈ 𝔖 ∧ (S_1 ∪ S_2) ∈ 𝔖
            ∧ S_1 ≺ S_2
    parse ('seeds', z_1, h_1) ← SS_{S_1}
    parse ('seeds', z_2, h_2) ← SS_{S_1}
    (z, c) ← TKDF.Derive(z_1, z_2)
    h ← H_pub(h_1, h_2, c)
    C ← (h_1, h_2, c)
    SS ← ('seeds', z, h)
    return (SS, C)

DeriveKS(S, SS_S)
    req S ∈ 𝔖
    if ∃e ∈ ℕ \ B : S = {e} then
        parse ('seeds', s, h) ← SS_S
        k ← TKDF.Key(s)
        KS_S ← ('keys', k, h)
    else
        parse ('seeds', z, h) ← SS_S
        t ← TKDF.Trapdoor(z)
        KS_S ← ('keys', t, h)
    return KS_S

Expand(S, KS_S, C_S, S_1, S_2)
    req S, S_1, S_2 ∈ 𝔖 ∧ (S_1 ∪ S_2) = S
            ∧ S_1 ≺ S_2
    parse ('keys', t, h) ← KS_S
    parse (h_1, h_2, c) ← C_S
    req h = H_pub(h_1, h_2, c)
    (t_1, t_2) ← TKDF.Invert(c, t)
    KS_1 ← ('keys', t_1, h_1)
    KS_2 ← ('keys', t_2, h_2)
    return (KS_1, KS_2)

Recover(e, KS_{e}, C_{e})
    if e ∈ B then
        parse ('keys', t, h) ← KS_{e}
        parse (h', ⊥, c)
        req h' = h ∧ h = H_pub(e)
        (k, t') ← TKDF.Invert(c, t)
        req t' = TKDF.Trapdoor(z̃)
    else
        parse ('keys', k, h) ← KS_{e}
        req h = H_pub(e)
    return k
```

Fig. 12: A description of the I-CKS scheme based on a (symmetric) TKDF. Note that for brevity we did not include the public hash key pub as part of every state. Furthermore, in Compact and Expand we assume $\mathcal{S}_1 \prec \mathcal{S}_2$; the general protocol is obtained by invoking the algorithm with reversed arguments in the other case.

Integrity. Finally, let us enhance our protocol to satisfy integrity. This can be done using a collision-resistant hash function $\mathrm{H}_{\mathrm{pub}}(\cdot)$. Each state is enhanced with a hash, where the hash of a combined state is set to $h = \mathrm{H}_{\mathrm{pub}}(h_1, h_2, \mathrm{c})$, when computing Compact on seed states with hashes h_1 and h_2, respectively, and c is the TKDF ciphertext produced during Compact. The hashes h_1 and h_2 are then output as part of the ciphertext along c, i.e., we set $C = (h_1, h_2, \mathrm{c})$. For an immediate state $SS_{\{\mathsf{e}\}}$ the hash just binds the epoch number e.

Correctness and security. The correctness of the scheme follows from the correctness of the TKDF and inspection.

Theorem 4. *The iterative CKS scheme from Fig. 12 is correct if the underlying TKDF is correct and has deterministic ciphertexts. More concretely,*

$$\mathsf{Adv}_{\mathsf{G}^{\mathrm{Corr}}_{\mathrm{I-CKS}}}(\mathcal{A}) \leq (q_{\mathrm{Init}} + q_{\mathrm{Compact}}) \cdot \mathsf{Adv}_{\mathsf{G}^{\mathrm{Corr}}_{\mathrm{TKDF}}}(\mathcal{A}'))$$

where q_{Init} and q_{Compact} denote bounds on the number of Init *and* Compact *calls.*

Proof. Note that the TKDF states and trapdoor are deterministic functions of the inputs. Therefore, each unique set $\mathcal{K} = \{(\mathsf{e}, \mathrm{s}_{\mathsf{e}}) \mid \mathsf{e} \in \mathcal{S}\}$ of epoch-seed pairs there is exactly one corresponding seed state $SS_{\mathcal{S}}$ and key state $KS_{\mathcal{S}}$. As a result, for each $\mathcal{K}$ there exists a unique TKDF state z_i and t_i. Finally not that $q_{\mathrm{Init}} + q_{\mathrm{Compact}}$ is a bound on the number of TKDF instances. Correctness is thus directly implied by the correctness of the TKDF. □

We now establish security of the TKDF scheme.

Theorem 5. *The iterative CKS scheme from Fig. 12 satisfies real-or-random security of keys if the TKDF is secure. More concretely,*

$$\mathsf{Adv}_{\mathsf{G}^{\mathrm{Keys\text{-}RoR}}_{\mathrm{I\text{-}CKS}}}(\mathcal{A}) \leq (q_{\mathrm{Init}} + q_{\mathrm{Compact}}) \cdot \big(\mathsf{Adv}_{\mathsf{G}^{\mathrm{KeyRand}}_{\mathrm{TKDF}}}(\mathcal{A}_1) + \mathsf{Adv}_{\mathsf{G}^{\mathrm{TdRand}}_{\mathrm{TKDF}}}(\mathcal{A}_2) + \mathsf{Adv}_{\mathsf{G}^{\mathrm{IND\text{-}CPA\text{-}OT}}_{\mathrm{TKDF}}}(\mathcal{A}_3)\big)$$

where q_{Init} and q_{Compact} denote bounds on the number of Init *and* Compact *calls.*

Proof. We perform two sequences of hybrids over $\mathcal{D}ag_{\mathfrak{S}}$. (Technically, over the subset of vertices the adversary explores during the interaction, i.e., the set of $\mathcal{S}$ for which $SS_{\mathcal{S}}$ is computed.) More specifically, we actually have one copy of the node for each distinct seed state $SS_{\mathcal{S}}$ the adversary generates for the node. In each hybrid step, exactly one node gets touched. For simplicity, we ignore the hashes in the following, as they are only computed on public data (ciphertexts and other hashes). We say that the node is secure if at least one of its children is secure. A source is said to be secure if it uses a seed not provided by the adversary and if its epoch is not in $\mathcal{C}orr$.

The first sequence of hybrids is in the direction of the graph, i.e., we visit each secure node once all its children have been visited. Consider a particular node characterized by $v = (\mathcal{S}, \mathrm{z}_{\mathcal{S}}, \mathrm{t}_{\mathcal{S}}, \mathrm{c}_{\mathcal{S}})$. Do the following steps:

– For all its parent nodes that have not been such that v is a left child (according to $\prec$), replace $t_{\mathcal{S}}$ by a freshly sampled value and $c_{\mathcal{S}}$ with a simulated ciphertext.
– Analogously for parent nodes where v is the right child. (Replace each node only once.)

Note that the invariant we maintain is that when starting the hybrid step, v already has an independent TKDF trapdoor and simulated ciphertext, meaning its z is independent of nodes lower in the graph. Therefore, the first step is indistinguishable by key randomness whereas the latter by trapdoor randomness. (For source nodes with $\mathsf{e} \in \mathcal{B}$ initially perform the same step.)

Now consider the second sequence of hybrids in the reverse direction. For simplicity, consider a sink $v = (\mathcal{S}, z_{\mathcal{S}}, t_{\mathcal{S}}, c_{\mathcal{S}})$. Observe that $z_{\mathcal{S}}$ is not used at all in the game. Therefore, we can apply semantic security to replace $c_{\mathcal{S}}$ with a ciphertext that is independent of all other values in the game. More generally, once for v the ciphertext of all its parent nodes have had their ciphertext replaced, the above argument holds.

At the end of the second hybrid sequence, all trapdoors/keys assigned to secure nodes are (1) independently and uniformly sampled and (2) not otherwise used in the game. It is easy to see that in this game the adversary can thus not distinguish "real" keys from random ones. □

Theorem 6. *The iterative CKS scheme from Fig. 12 satisfies integrity if the hash function* $\mathrm{H_{pub}}(\cdot)$ *is collision-resistant and the TKDF correct and has deterministic ciphertexts. More concretely,*

$$\mathsf{Adv}_{\mathsf{G}^{\text{Integrity}}_{\text{I-CKS}}}(\mathcal{A}) \leq (q_{\text{Init}} + q_{\text{Compact}}) \cdot \left(\mathsf{Adv}_{\mathsf{G}^{\text{Corr}}_{\text{TKDF}}}(\mathcal{A}') + \mathsf{Adv}_{\mathsf{G}^{\text{CR}}_{\text{H}}}(\mathcal{A}'')\right)$$

where q_{Init} *and* q_{Compact} *denote bounds on the number of* Init *and* Compact *calls.*

Proof. By collision resistance of the hash function $\mathrm{H_{pub}}(\cdot)$, Expand and Recover will only accept the exact ciphertext C used to generate the respective keys space. Furthermore, the protocol will only accept the correct hash values for the child nodes in $\mathcal{D}ag_{\mathfrak{S}}$ as well. This reduces integrity to correctness, which in turn is implied by the correctness of the TKDF. □

Variants. The above scheme is secure against an active adversary controlling the server (i.e., delivering wrong ciphertexts) and malicious insiders making parties use inconsistent and adversarially chosen seeds. We state some simple observations about weaker security models. First, we note that hash functions are only needed to protect against an active attacker delivering wrong ciphertexts.

Corollary 3. *Standard-model I-CKS against honest-but-curious servers, i.e., without integrity, can be built from dual-PRFs. Additionally, the TKDF ciphertexts do not need to be deterministic.*

Second, we observe that one-time TKDF security suffices when considering outsider security only.

Lemma 1. *Outsider-secure I-CKS, e.g., when restricting the adversary to submit* $s_{\mathsf{e}} = \perp$ *for the* Init *oracle can be built from one-time secure TKDF and, therefore, from one-way function only.*

Proof. Observe that in the proof of real-or-random security, the number of $\mathrm{TKDF.Derive}(z, \cdot)$ queries correspond to the number of distinct other seed states z gets compacted with. In the case of outsider security, those states are unique (for each set $\mathcal{S}$). Therefore, one-time key randomness suffices. The analogous argument can be made for $\mathrm{TKDF.Derive}(\cdot, z)$ and one-time trapdoor randomness.

5 CKS from Iterative CKS

In this section, we sketch how to turn any iterative CKS scheme into a regular (standard model) CKS scheme. This provides a template for how to use iterative CKS for a group of parties to outsource a sequence of keys that are derived from seeds. Note that the key derivation $\mathrm{U.Key}(\mathsf{e}, s)$ for the standard-model CKS scheme is just the one from the iterative CKS scheme, i.e., $\mathrm{DeriveKey}(s_{\mathsf{e}}, \mathsf{e})$. If we target insider security or security against malicious servers, we require the iterative CKS scheme to have deterministic ciphertexts. For the weakest passive security notion, any I-CKS scheme suffices. In the following, we consider the stronger security notion (and briefly discuss the weaker variants).

Server state and algorithms. The server just implements a bulletin board BB. Each entry stores an I-CKS ciphertext C and is indexed by a collision-resistant hash thereof, i.e., $\mathrm{BB}[\mathrm{H_{pub}}(C)] = C$. Importantly, the server will compute the hash themselves. This ensures that a malicious insider cannot overwrite a valid ciphertext of another user, or preemptively set a position of the bulletin board to something invalid. For a passively secure CKS protocol, we can index the bulletin board using a description of the set $\mathfrak{S}$ instead, i.e., $\mathrm{BB}[\mathcal{S}] = C_{\mathcal{S}}$. The server will then store the first ciphertext sent for each $\mathcal{S}$ and ignore all subsequent ones.

As we will see, U.Upload will just send a set of ciphertexts that the server will store. Similarly, U.Erase, U.Grant, U.Accept, and U.retrieve will query the bulletin board for a subset of positions. We note that for our specific iterative CKS scheme, this could be a bit optimized: as $C = (h_1, h_2, c)$ the user would not need to (iteratively) query for h_1 and h_2 and the server could just (recursively) include all ciphertexts needed by the protocol.

User state. The client state depends on the graph $\mathcal{D}ag_{\mathfrak{S}}$. More specifically, the client stores the seeds state $SS_{\mathcal{S}}$ or the keys state $KS_{\mathcal{S}}$ for a subset of nodes. We say that for a node $\mathcal{S} \in \mathfrak{S}$, we say that $SS_{\mathcal{S}}$ is *derivable* if either (a) $SS_{\mathcal{S}}$ is directly stored or (b) it is derivable for at least one of the parent nodes of $\mathcal{S}$. We say that $KS_{\mathcal{S}}$ is derivable if either (a) $KS_{\mathcal{S}}$ is stored, (b) $SS_{\mathcal{S}}$ is stored, or (c) the keys state is derivable for at least one of the parent nodes of $\mathcal{S}$. We say that $SS_{\mathcal{S}}$ and $KS_{\mathcal{S}}$ are *indirectly* derivable if the respective options (a) do not apply. The user state is then maintained subject to the following invariant, where additional seeds states or keys states are purged.

Invariant 1 (Compactness) *A node $\mathcal{S} \in \mathfrak{S}$ only has the seeds state SS stored if it's not indirectly derivable. Analogously, it only has the keys state KS stored if it's not indirectly derivable. (This in particular means that that no node stores both seeds and keys state.)*

Appending seeds. When appending a seed s_{e} for an epoch e, the algorithm proceeds in three steps:

1. Create a seeds state $SS_{\{\mathsf{e}\}} \leftarrow \mathrm{Init}(\mathsf{e}, \mathrm{s}_{\mathsf{e}})$ for the leaf node.
2. Iteratively derive seeds along all paths starting at the leaf node using Compact. That is, for any of the nodes along a path, if the seeds states of both children are known, then use Compact to compute the one for this node.
3. Purge any seeds states or keys states that violate compactness.

The upload state $\mathrm{st}_{\mathrm{up}}$ then contains all ciphertexts produced by Compact. We assume those ciphertexts to be deterministic (which does not violate security; for instance, our TKDF uses a one-time IND-CPA secure encryption scheme) then they can simply be sent to the server.

Retrieving keys. To retrieve the key k_{e} for an epoch e, U.retrieve identifies a node on a path from the leaf $\mathcal{S} = \{\mathsf{e}\}$ for which either the seeds state or the keys state is stored. (If multiple candidates exist, pick e.g. the one the shortest distance from the leaf.) If it is a seeds state, use DeriveKS to derive the respective keys state. Then the algorithm uses Expand and finally Recover to retrieve the key. For each step, request the necessary ciphertexts from the server. To retrieve the keys for an entire subset of epochs $\mathrm{share} \subset \mathbb{N}$, the above steps are generalized by identifying a suitable set of (internal) nodes that cover share with respect to reachability in $\mathcal{D}ag_{\mathfrak{S}}$.

Delegation. Delegation works similarly to the retrieval of keys, except that internal nodes are not further expanded if all or their descendants are part of the set of delegated keys $\mathrm{share} \subset \mathbb{N}$. Note that the delegation message can either contain keys states $KS_{\mathcal{S}}$ or $SS_{\mathcal{S}}$. The latter is preferable if it is stored by the delegating user. The accepting user receives those elements. If they already know keys for any epoch $\mathsf{e} \in \mathrm{share}$ they check consistency by recovering the key k_{e} according to their own state and the retrieved one. Afterward, they add the obtained information to their local state and compact it.

Erasing keys. Erasure works like self-delegation of the epochs not erased, except that no consistency checks are needed. That is, if a user knows — i.e., can currently recover — keys for epochs $\mathcal{K}$ and wants to erase share, then they self-delegate $\mathcal{K} \setminus \mathrm{share}$.

Functionality and efficiency. Observe that the efficiency of the above scheme inherently depends on $\mathcal{D}ag_{\mathfrak{S}}$. As a result, the choice of $\mathcal{D}ag_{\mathfrak{S}}$ also dictates which sets of epochs can be efficiently retrieved, delegated, and erased as formalized by $\mathcal{R}$, $\mathcal{G}$, and $\mathcal{E}$. We do not make this connection fully formal but only highlight some of the relations.

- *Small covers.* For the user state to be compact, there need to exist nodes that "cover" large sets of epochs, i.e., for which a large set of leaf nodes are descendants. For instance in the all-or-nothing scheme in [7] there exist nodes that cover $[1, n]$, for any n, and in the interval scheme nodes that cover $[2^i, 2^i + 2^j - 1]$ for $i, j \in \{0, 1, \ldots\}$.
  Similarly, delegation and erasure only work efficiently for subsets $\mathrm{share} \subset \mathbb{N}$ that have a small cover, sub-linear in share.
- *Limited out degrees.* If a node in $\mathcal{D}ag_{\mathfrak{S}}$ has too many ancestors on disjoint paths, then U.Append becomes inefficient. In [7], both schemes used a structure where each node has out-degree 1.
- *Short diameter.* If $\mathcal{D}ag_{\mathfrak{S}}$ contains too long path, then U.Append or U.retrieve can become inefficient. This is, for instance, the reason why the all-or-nothing scheme cannot support efficient (i.e., sub-linear) retrieval of individual keys.

Security and correctness. The security of the standard-model CKS scheme reduces directly to the respective properties of the iterative CKS scheme. In other words, the integrity of the CKS scheme follows from the integrity of the I-CKS scheme, and analogously for pseudorandomness and correctness. The proof of the following theorem follows by inspection.

Theorem 7 (Informal). *The above sketched CKS scheme is correct and secure if the iterative CKS scheme is correct and secure. The same furthermore applies to the variants considering passive security or security against an honest-but-curious server.*

Legacy compatibility. Observe that the result from Theorem 3 carries over to the entire CKS scheme.

Corollary 4. *Assume there is an E2E-secure application that provides a secure KDF, that we can evaluate on one additional input, to derive keys from initial seeds (and does not otherwise use the seeds). Then we can build a CKS scheme that is compatible with said application that recovers the keys.*

In particular, this class of legacy applications contains common schemes such as the Double Ratchet or MLS, which use a key schedule based on HKDF.

References

1. Bagherzandi, A., Jarecki, S., Saxena, N., Lu, Y.: Password-protected secret sharing. In: Chen, Y., Danezis, G., Shmatikov, V. (eds.) ACM CCS 2011. pp. 433–444. ACM Press (Oct 2011). https://doi.org/10.1145/2046707.2046758
2. Bellare, M.: New proofs for NMAC and HMAC: Security without collision resistance. Journal of Cryptology **28**(4), 844–878 (Oct 2015). https://doi.org/10.1007/s00145-014-9185-x
3. Bellare, M., Keelveedhi, S., Ristenpart, T.: Message-locked encryption and secure deduplication. In: Johansson, T., Nguyen, P.Q. (eds.) EUROCRYPT 2013. LNCS, vol. 7881, pp. 296–312. Springer, Heidelberg (May 2013). https://doi.org/10.1007/978-3-642-38348-9_18
4. Bellare, M., Lysyanskaya, A.: Symmetric and dual PRFs from standard assumptions: A generic validation of an HMAC assumption. Cryptology ePrint Archive, Report 2015/1198 (2015), https://eprint.iacr.org/2015/1198
5. Brzuska, C., Cornelissen, E., Kohbrok, K.: Security analysis of the MLS key derivation. In: 2022 IEEE Symposium on Security and Privacy. pp. 2535–2553. IEEE Computer Society Press (May 2022). https://doi.org/10.1109/SP46214.2022.9833678
6. Das, P., Hesse, J., Lehmann, A.: DPaSE: Distributed password-authenticated symmetric-key encryption, or how to get many keys from one password. In: Suga, Y., Sakurai, K., Ding, X., Sako, K. (eds.) ASIACCS 22. pp. 682–696. ACM Press (May / Jun 2022). https://doi.org/10.1145/3488932.3517389
7. Dodis, Y., Jost, D., Marcedone, A.: Compact key storage. In: Reyzin, L., Stebila, D. (eds.) Advances in Cryptology – CRYPTO 2024. pp. 75–109. Springer Nature Switzerland, Cham (2024)
8. Douceur, J., Adya, A., Bolosky, W., Simon, P., Theimer, M.: Reclaiming space from duplicate files in a serverless distributed file system. In: Proceedings 22nd International Conference on Distributed Computing Systems. pp. 617–624 (2002). https://doi.org/10.1109/ICDCS.2002.1022312
9. Fábrega, A., Pérez, C.O., Namavari, A., Nassi, B., Agarwal, R., Ristenpart, T.: Injection attacks against end-to-end encrypted applications. In: 2024 IEEE Symposium on Security and Privacy (SP). pp. 85–85. IEEE Computer Society, Los Alamitos, CA, USA (may 2024). https://doi.org/10.1109/SP54263.2024.00082, https://doi.ieeecomputersociety.org/10.1109/SP54263.2024.00082
10. Jafargholi, Z., Kamath, C., Klein, K., Komargodski, I., Pietrzak, K., Wichs, D.: Be adaptive, avoid overcommitting. In: Katz, J., Shacham, H. (eds.) CRYPTO 2017, Part I. LNCS, vol. 10401, pp. 133–163. Springer, Heidelberg (Aug 2017). https://doi.org/10.1007/978-3-319-63688-7_5
11. Jarecki, S., Kiayias, A., Krawczyk, H., Xu, J.: Highly-efficient and composable password-protected secret sharing (or: How to protect your bitcoin wallet online). In: 2016 IEEE European Symposium on Security and Privacy (EuroS&P). pp. 276–291. IEEE Computer Society, Los Alamitos, CA, USA (mar 2016). https://doi.org/10.1109/EuroSP.2016.30, https://doi.ieeecomputersociety.org/10.1109/EuroSP.2016.30
12. Jarecki, S., Krawczyk, H., Resch, J.K.: Updatable oblivious key management for storage systems. In: Cavallaro, L., Kinder, J., Wang, X., Katz, J. (eds.) ACM CCS 2019. pp. 379–393. ACM Press (Nov 2019). https://doi.org/10.1145/3319535.3363196
13. Kamath, C., Klein, K., Pietrzak, K., Walter, M.: The cost of adaptivity in security games on graphs. In: Nissim, K., Waters, B. (eds.) TCC 2021, Part II. LNCS, vol. 13043, pp. 550–581. Springer, Heidelberg (Nov 2021). https://doi.org/10.1007/978-3-030-90453-1_19
14. Krawczyk, H.: Cryptographic extraction and key derivation: The HKDF scheme. In: Rabin, T. (ed.) CRYPTO 2010. LNCS, vol. 6223, pp. 631–648. Springer, Heidelberg (Aug 2010). https://doi.org/10.1007/978-3-642-14623-7_34

A Preliminaries

Notation. Let $\mathbb{N} := \{1, 2, \ldots\}$. $x \leftarrow a$ denotes assigning the value a to the variable x and, for a set $\mathcal{S}$, we write $x \leftarrow\$ \mathcal{S}$ to denote sampling an element uniformly at random. $\mathcal{P}(\mathcal{S})$ denotes the powerset of $\mathcal{S}$ and $\kappa \in \mathbb{N}$ the security parameter. When describing stateful algorithms or security games, we make use of the following special keywords. First, for a boolean condition cond, the statement **req** cond is a shorthand for "if cond is false, revert all changes to the state made during this invocation and return an error $\perp$." Second, the statement **parse** $(x, y) \leftarrow z$ denotes the attempt to parse z as a tuple and abort analogous to **req** in case this is not possible. Third, we use **try** $y \leftarrow \mathsf{A}(x)$ to denote that if the invocation of algorithm A fails, the calling procedure itself unwinds and aborts with an error.

Interactive Algorithms. For describing interactive algorithms, we adapt the notation of [7] and assume a simple notion of (token-based) interactive algorithms. An execution of a two-party algorithm Alg between parties A and B is denoted $(y_A; y_B) \leftarrow \langle A.\mathsf{Alg}(x_A) \leftrightarrow B.\mathsf{Alg}(x_B)\rangle$, where x_A and x_B denote the respective parties' inputs and y_A and y_B their outputs. In particular, we view oracle machines as a special case of such an interactive algorithm where the oracle name is appropriately encoded as part of a message sent from the invoking party to the invoked party.

When executing an interactive protocol between two honest parties in a security game we assume that the oracle blocks until the interaction is finished. We write $(y_U; \perp) \leftarrow \langle U.\mathsf{Alg}(x_U) \leftrightarrow \mathcal{A}\rangle$ to denote that the adversary acts as the second party. In that case, whenever the interactive protocol transfers control to the adversary, they may either choose to reply and transfer control back to the party U, or may choose to invoke a different oracle, or answer to a different ongoing interactive protocol. As such, the adversary may arbitrarily interleave oracle invocations and protocol executions (unless otherwise specified). The execution of the oracle code continues once the honest party terminates.

Cryptographic Primitives. We use some basic cryptographic primitives.

Pseudo-Random Generators. A length-doubling pseudo-random generator $\mathrm{PRG}\colon \{0,1\}^\kappa \to \{0,1\}^{2\kappa}$. We require that

$$\left\{\mathrm{PRG}(U_\kappa)\right\}_{\kappa\in\mathbb{N}} \approx_c \left\{U_{2\kappa}\right\}_{\kappa\in\mathbb{N}}$$

where U_n denotes the uniform distribution over $\{0,1\}^n$.

Dual-PRF. A dual-PRF, as introduced in [2], is a function $\mathrm{dPRF}\colon \mathcal{X} \times \mathcal{Y} \to \mathcal{Z}$ is a deterministic algorithm such that for uniform random $x \leftarrow\$ \mathcal{X}$ and $y \leftarrow\$ \mathcal{Y}$, both $\mathrm{dPRF}(x, \cdot)$ and $\mathrm{dPRF}(\cdot, y)$ behave like a PRF. That is, the former is computationally indistinguishable, for any PPT adversary $\mathcal{A}$ having oracle access, from a uniform random function $F\colon \mathcal{Y} \to \mathcal{Z}$ and the latter form a uniform random function $G\colon \mathcal{X} \to \mathcal{Z}$.

Hash functions. We use a family of hash functions $\mathrm{H}_k(\cdot)\colon \{0,1\}^* \to \{0,1\}^\kappa$, indexed by a public hash-key $k \in \{0,1\}^\kappa$. We require the function family to be collision-resistant, namely, that given k chosen uniformly at random any PPT adversary $\mathcal{A}$ has negligible probability in finding $x \neq x'$ such that $\mathrm{H}_k(x) = \mathrm{H}_k(x')$.

Symmetric encryption. Finally, we require a one-time IND-CPA secure symmetric encryption scheme $\mathrm{SE} = (\mathrm{SE.Enc}, \mathrm{SE.Dec})$. Here one-time security means that the adversary only gets to make a single challenge (m_0, m_1) to the challenger — to receive an encryption of m_b — with no further encryption (or decryption) queries allowed.

B Standard-model CKS: Definitions

In this section, we formally define correctness and integrity for standard-model CKS. The definitions follow closely the ones for ROM-CKS as introduced in [7].

B.1 Correctness

In a nutshell, correctness requires that parties retrieve the correct keys when interacting with an honest server. Note that the correctness notion accounts for malicious insiders: even when a malicious insider uploads information to the honest server, correctness must be ensured for honest users. The game is a rather straightforward modification of the one from [7], the main differences are:

- The game keeps track separately of for which epoch a user u knows the seed and key using the $\mathsf{KnownSeed}$ and $\mathsf{KnownKey}$ arrays, respectively, instead of using a single Known array. Whenever a seed is appended, the user is assumed to both know the seed and key, whereas delegation is only assumed to delegate keys.
- Accordingly, the game tracks the concrete seeds and keys each user knows using $\mathsf{Seeds}[\mathsf{u}, \mathsf{e}]$ and $\mathsf{Keys}[\mathsf{u}, \mathsf{e}]$. The latter is now used for all consistency checks, i.e., the game uses $\mathsf{Keys}[\mathsf{u}, \mathsf{e}]$ to ensure that a user never retrieves an inconsistent key. The $\mathsf{Seeds}[\mathsf{u}, \mathsf{e}]$ is used for functionality as part of `subsumed`. More concretely, for correctness, we say that a party must upload unless another party knows at least as many *seeds*. This models that a party only having been delegated keys might not be able to contribute to the shared CKS state in the same manner as a party having appended the underlying seed.

Game $(\mathcal{G}, \mathcal{R}, \mathcal{E})$-CKS-$\mathsf{Corr}^{\mathcal{A}}_{\mathsf{CKS}}$ (Correctness)

```
Main
  win ← false, n ← 0
  St[·], Seeds[·,·], Keys[·,·], GrantInfo[·] ← ⊥
  KnownSeed[·,·], KnownKey[·,·] ← false
  st_S ← S.Init(1^κ)
  A^{CreateUser,...,UploadAdv}(1^κ, st_S)
  return win

Oracle CreateUser
  n ← n + 1
  St[n] ← U.Init(1^κ)
  return St[n]

Oracle Append
Input: (u, e, s, upload) ∈ [n] × ℕ × {0,1}^κ × {0,1}
  req Keys[u, e] = ⊥
  Seeds[u, e] ← s; Keys[u, e] ← U.Key(e, s)
  KnownSeed[u, e] ← true; KnownKey[u, e] ← true
  if ¬subsumed(u) then
    upload ← true
  (St[u], st_up) ← U.Append(St[u], e, s, upload)
  if upload then
    (⊥; st_S) ← ⟨U.Upload(st_up) ↔ S.Upload(st_S)⟩
  return (st_S, St[u], st_up)

Oracle Erase
Input: (u, share) ∈ [n] × P(ℕ)
  req E(KnownSeed[u, ·], KnownKey[u, ·], share)
  (st′; ⊥) ← ⟨U.Erase(St[u], share) ↔ S.Erase(st_S)⟩
  if st′ = ERROR then
    if ∃e ∈ share : Keys[u, e] ≠ ⊥ then win ← true
  else
    St[u] ← st′
    for e ∈ share do
      KnownSeed[u, e] ← false
      KnownKey[u, e] ← false
      Seeds[u, e] ← ⊥; Keys[u, e] ← ⊥
  return st′

Oracle Retrieve
Input: (u, share) ∈ [n] × P(ℕ)
  req R(KnownSeed[u, ·], KnownKey[u, ·], share)
  (keys; ⊥) ←
      ⟨U.Retrieve(St[u], share) ↔ S.Retrieve(st_S)⟩
  if keys = ERROR then
    if ∀e ∈ share : Keys[u, e] ≠ ⊥ then win ← true
  else
    for e ∈ share do
      if Keys[u, e] ∉ {⊥, keys(e)} then
        win ← true
      Keys[u, e] ← keys(e)
  return keys

Oracle Grant
Input: (u, share) ∈ [n] × [n] × P(ℕ)
  req G(KnownSeed[u, ·], KnownKey[u, ·], share)
  (msg; ⊥) ← ⟨U.Grant(St[u], share) ↔ S.Grant(st_S)⟩
  if msg = ERROR then
    if ∀e ∈ share : Keys[u, e] ≠ ⊥ then win ← true
  else
    keys[·] ← ⊥
    for e ∈ share do
      keys[e] ← Keys[u, e]
    GrantInfo[msg] ← (share, keys)
  return msg

Oracle Accept
Input: (u′, msg, share, upload)
         ∈ [n] × {0,1}* × P(ℕ) × {0,1}
  (share′, keys) ← GrantInfo[msg]
  if GrantInfo[msg] ≠ ⊥ ∧ share′ = share then
    for e ∈ share do
      Keys[u, e] ← keys[e]
      KnownKey[u, e] ← true
    if ¬subsumed(u′) then upload ← true
  else upload ← true
  (st′, st_up; ⊥) ← ⟨U.Accept(St[u′], share, msg, upload)
                       ↔ S.Accept(st_S)⟩
  if upload then
    (⊥; st_S) ← ⟨U.Upload(st_up) ↔ S.Upload(st_S)⟩
  if GrantInfo[msg] ≠ ⊥ ∧ share′ = share then
    if st′ = ERROR then win ← true
    St[u′] ← st′
  else if st′ ≠ ERROR then
    for e ∈ share do Known[u, e] ← true
    St[u′] ← st′
  return (st_S, st′)

Oracle UploadAdv
  (⊥; st_S) ← ⟨A ↔ S.Upload(st_S)⟩
  return st_S

subsumed
Input: u ∈ [n]
  return ∃u′ ≠ u ∈ [n] :
           ∀e : Seeds[u, e] ∈ {⊥, Seeds[u′, e]}
```

Fig. 13: The Compact Key Storage correctness notion. We assume that the adversary does not interleave calls to UploadAdv with calls to other oracles.

– The retrieval predicate $\mathcal{R}$, delegation predicate $\mathcal{G}$, and erasure predicate $\mathcal{E}$ are evaluated on both KnownSeed and KnownKey, in correspondence to their generalized definition for standard-model CKS.

Let us briefly discuss some aspects of the correctness game — we refer to [7] for a more in-depth discussion. First, observe the use of the upload flag in the Append and Accept oracles. Using this flag, the adversary can force the user to run the upload procedure, as correctness must hold even if multiple users upload. The flag is overwritten in case the user must upload for correctness, as determined by the `subsumed` helper. Second, observe the slight discrepancy between $\mathsf{KnownKey}[u, e]$ and $\mathsf{Keys}[u, e]$. In principle, whenever a user has a concrete key assigned, i.e., $\mathsf{Keys}[u, e] \neq \perp$, then $\mathsf{KnownKey}[u, e] = \mathtt{true}$. The converse does not necessarily hold, however. If the user is granted some key by a malicious user (indicated by msg not produced by the Grant oracle) then the game does not know the resulting key. $\mathsf{Keys}[u, e]$ is then set whenever the game first learns of the injected key, to ensure that the user only outputs one key for a given epoch even after a malicious delegation. Finally, observe the error handling. Erase, Retrieve, and Grant may all fail after having accepted a malicious grant. This allows Accept to be efficient and not verify the entire set of keys share upfront. Erase is allowed to fail if *all* keys where maliciously injected; therefore any secure key can be erase. Retrieve and Grant may even fail if at least one of the involved keys has been injected.

Definition 13. *We say that a $(\mathcal{G}, \mathcal{R}, \mathcal{E})$-CKS scheme* CKS *is correct, if the probability of any adversary $\mathcal{A}$ winning the $(\mathcal{G}, \mathcal{R}, \mathcal{E})$-CKS-$\mathsf{Corr}^{\mathcal{A}}_{\mathsf{CKS}}$ game from Fig. 13 is negligible in κ.*

B.2 Integrity

Integrity requires that users either restore their correct key, i.e., the one they outsourced, or output an explicit error indicating that a key could not be restored. The modifications compared to ROM-CKS are similar to the one for the correctness notion. In particular, the game now keeps track of the keys a user outsources instead of the seeds.

Overall, the game works fairly similarly to the correctness game. Let us briefly mention some of the intricacies. First, note that the game uses placeholders for keys not yet known to the game. Concretely, whenever a user u accepts an injected grant message msg, the game assigns placeholders $\mathfrak{K}_p$, for $p \in \mathbb{N}$, for all the keys. Those placeholders are later (globally) replaced once the key becomes known. This allows to enforce consistency even if u further delegates those keys. Second, observe that all algorithms are now allowed to fail, as indicated with the **try** keyword. Integrity guarantees consistency only whenever users succeed. Finally, we note that the game formalizes a strong variant of integrity where the adversary is assumed to know all secret states. This guarantees that even state leakages cannot break integrity.

Definition 14. *We say that a CKS scheme* CKS *satisfies integrity, if the probability of any adversary $\mathcal{A}$ winning the* CKS-$\mathsf{Int}^{\mathcal{A}}_{\mathsf{CKS}}$ *game from Fig. 14 is negligible in κ.*

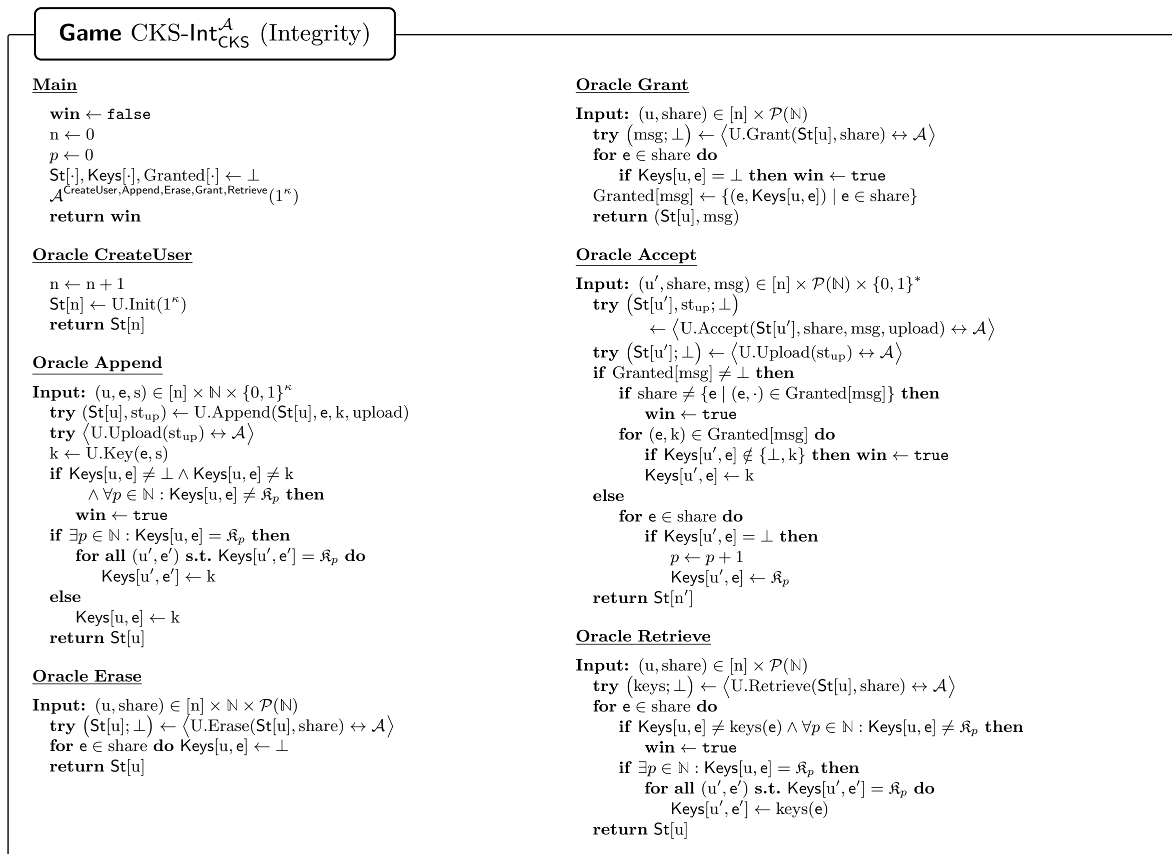


Fig. 14: The integrity notion of CKS. The values $\mathfrak{K}_p$, for $p \in \mathbb{N}$, are placeholders that are assumed to be distinct symbols, i.e., $\mathfrak{K}_p \neq \perp$, $\mathfrak{K}_p \neq k$ for any bitstring k, and $\mathfrak{K}_p \neq \mathfrak{K}_q$ for $p \neq q$.