Compact Key Storage
A Modern Approach to Key Backup and Delegation*

Yevgeniy Dodis[1†], Daniel Jost[1‡], and Antonio Marcedone[2]

[1] New York University {dodis, daniel.jost}@cs.nyu.edu
[2] Zoom Video Communications antonio.marcedone@zoom.us

Abstract. End-to-End (E2E) encrypted messaging, which prevents even the service provider from learning communication contents, is gaining popularity. Since users care about maintaining access to their data even if their devices are lost or broken or just replaced, these systems are often paired with cloud backup solutions: Typically, the user will encrypt their messages with a fixed key, and upload the ciphertexts to the server. Unfortunately, this naive solution has many drawbacks. First, it often undermines the fancy security guarantees of the core application, such as forward secrecy (FS) and post-compromise security (PCS), in case the single backup key is compromised. Second, they are wasteful for backing up conversations in large groups, where many users are interested in backing up the same sequence of messages.

Instead, we formalize a new primitive called *Compact Key Storage* (CKS) as the "right" solution to this problem. Such CKS scheme allows a mutable set of parties to delegate to a server storage of an increasing set of keys, while each client maintains only a small state. Clients update their state as they learn new keys (maintaining PCS), or whenever they want to forget keys (achieving FS), often without the need to interact with the server. Moreover, access to the keys (or some subset of them) can be efficiently delegated to new group members, who all efficiently share the same server's storage.

We carefully define syntax, correctness, privacy, and integrity of CKS schemes, and build two efficient schemes provably satisfying these notions. Our *line scheme* covers the most basic "all-or-nothing" flavor of CKS, where one wishes to compactly store and delegate the entire history of past secrets. Thus, new users enjoy the efficiency and compactness properties of the CKS only after being granted access to the entire history of keys. In contrast, our *interval scheme* is only slightly less efficient but allows for finer-grained access, delegation, and deletion of past keys.

* This is the full version of an article with the same title appearing in the proceedings of CRYPTO 2024, Springer, © IACR 2024.

† Research partially conducted while contracting for Zoom. Research partially supported by NSF grant CNS-2055578, and gifts from JP Morgan, Protocol Labs, Stellar, and Algorand Foundation.

‡ Research partially conducted while contracting for Zoom.

Table of Contents

1 Introduction . . . 3
  1.1 Traditional Backup . . . 3
  1.2 Our Work: Compact Key Storage . . . 5
  1.3 Scheme Overview . . . 7
  1.4 Related Work . . . 9
  1.5 Outline . . . 10
2 Preliminaries . . . 10
3 Compact Key Storage . . . 10
  3.1 High-level CKS Overview . . . 10
  3.2 Syntax Definition . . . 11
  3.3 Correctness . . . 13
4 CKS Security . . . 13
  4.1 Definitional Challenges . . . 13
  4.2 Impossibility of Key-Indistinguishability . . . 16
  4.3 Preservation Security . . . 16
  4.4 Integrity . . . 20
5 Compact Key Storage Schemes . . . 20
  5.1 The Line Scheme . . . 21
  5.2 The Interval Scheme . . . 24
A Additional Preliminaries . . . 30
B On CKS-compatible Games . . . 30
  B.1 Single to Many Instances . . . 30
  B.2 CKS Compatibility of Cryptographic Primitives . . . 31
  B.3 CKS Compatibility of Signal . . . 34
C Details on CKS Security . . . 35
D Convergent Encryption . . . 36
E Details on the Line Scheme . . . 38
  E.1 A Formal Description of the Scheme . . . 38
  E.2 Security . . . 38
F Details on the Interval Scheme . . . 43
  F.1 A Formal Description of the Scheme . . . 43
  F.2 Security . . . 46

1 Introduction

Existing cryptographic literature on secure messaging heavily focuses on the core functionality: sending secure messages and the intermediate step of establishing shared key material. The security guarantees of the respective protocols, such as Signal's Double Ratchet [42,2] or the IETF MLS protocol [8,3] have been investigated in depth, formalizing and proving advanced properties such as *forward secrecy* (FS) and *post-compromise security* (PCS). This means that compromising internal secrets used by these messaging applications does not hurt the security of past (FS) and future (PCS) messages.

Bigger Picture and Our Main Question. The secure messaging protocols above are almost always just part of a bigger messaging application providing additional functionality. Most importantly for our purposes, the conversation history is often a valuable resource that is too prized to only store on one device. In particular, users do not want to lose their conversation history forever in case the device is replaced, lost, or stolen. In practice, this means that the application should also offer some *cloud backup* option to restore prior messages.

In a related vein, when a new user joins a group in a secure group messaging protocol such as MLS, FS (cryptographically) ensures that the past messages are not automatically given to the user (even if the user recorded prior ciphertexts before joining the group). In many business application scenarios, however, the need for a new member to know the conversation history of the team trumps the need to achieve FS. Thus, once again a special (often cryptographic) *delegation* method needs to ensure that new users are given prior history.

In both of these examples, the attacker would get extra information compared to the one provided in the analysis of the "secure messaging" component. For example, in the case of cloud backup, it would include the information stored in the cloud. More importantly, upon compromise, the attacker might get (at least some of) the "cloud access" credentials that the user needs for accessing the backup, in addition to the cryptographic material used (and analyzed) by the secure messaging application. Somewhat surprisingly, we argue below that in many cases such an attacker will be able to *completely break FS/PCS security* that the secure messaging application worked so hard to achieve. By this, we do not mean a "trivial break" where the attacker gets legitimate messages that the user should get anyway. Instead, the attacker would get (a) messages that the legitimate user thought were "securely erased," and; (b) future messages not yet sent or even generated, at the time the compromise occurs.

Breaking FS/PCS is not merely a theoretical concern and we argue that advising privacy-concerned users to turn off cloud backup is not a satisfying solution. For example, a user may wish to have their messages regularly backed up in general, or simply use an application that defaults to such a setting. However, if at some point they sent or received a particularly sensitive message, they may decide to erase *even after the automatic backup process has been run.* A user may also be coerced by law enforcement/immigration control to hand over the secrets to their cloud backup, or their device might simply be hacked. The ability to erase messages or restore security after a compromise, without entirely restarting the backup process, seems like a natural and possibly critical requirement.

As the main question of this work, we ask whether this state of affairs is inherent, and argue that *there is a better way to add cloud backup and/or delegation capabilities to secure messaging applications.* Before introducing our proposed solution, which we call *Compact Key Storage* (CKS), let us dive into the problem more closely, for concreteness focusing on the cloud backup scenario. But an interested reader can jump to Figs. 1 and 2 for a sneak peek of our approach.

1.1 Traditional Backup

Traditionally, backup has been considered an orthogonal problem to the core secure messaging (SM) application. That is, one only backs up the *plaintext messages* m_i produced by a SM application — any ciphertext $\tilde{c}_i$ or secret s_i used by the SM application is treated as an implementation detail[1] not

[1] For clearer comparison with our approach, we explicitly put the "Encryption Box" Enc as the place where messages are encrypted with fresh keys. While not entirely without loss of generality, this approach is

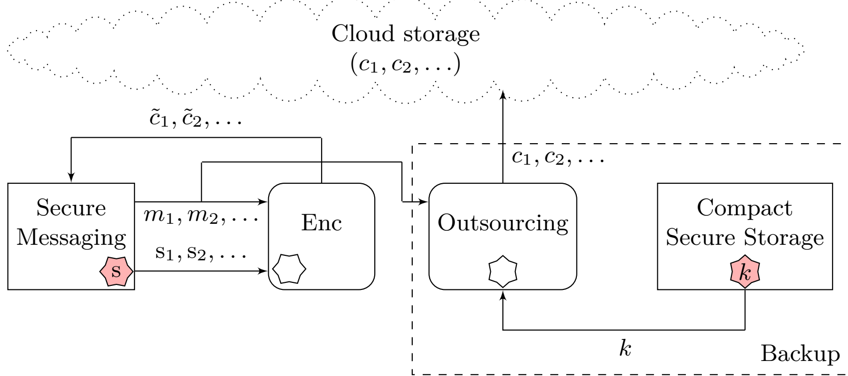

Fig. 1: A schematic representation of a classical cloud-backup process. The plaintext messages $m_1, m_2, \ldots$ are backed up independently of the secure-messaging scheme, secured by a static key k kept safe in a "Compact Secure Storage". Notice:
(1) the CCS secret k does not have PCS/FS unlike SM secrets $\mathrm{s}_1, \mathrm{s}_2, \ldots$;
(2) SM ciphertexts $\tilde{c}_i$ are not used for Backup, wasting storage & computation;
(3) fresh backup key k per user precludes deduplication.

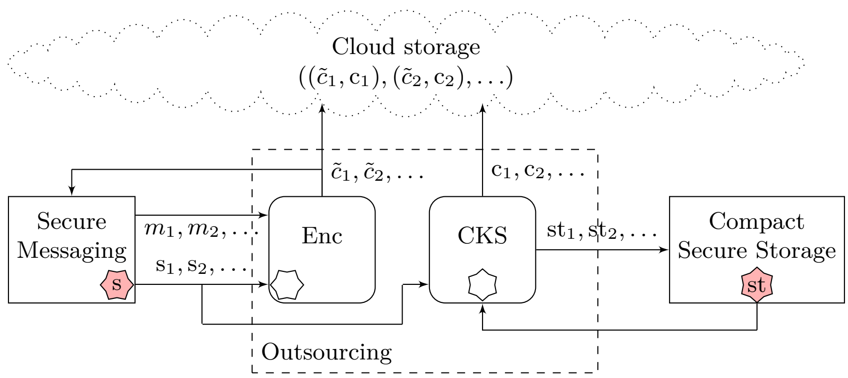

Fig. 2: A schematic representation of the CKS approach for backup. The secure-messaging ciphertexts $\tilde{c}_1, \tilde{c}_2, \ldots$ are stored and the secure-messaging secrets $\mathrm{s}_1, \mathrm{s}_2, \ldots$ are backed up. Notice:
(1) the CSS secret state st_i can offer PCS/FS analogous to SM secrets $\mathrm{s}_1, \mathrm{s}_2, \ldots$;
(2) reusing SM ciphertexts $\tilde{c}_i$ saves computation and enables shared storage;
(3) CKS state st is a deterministic function of $\mathrm{s}_1, \mathrm{s}_2, \ldots$, enabling deduplication.

exposed to the backup process. In fact, this is considered advantageous for the sake of modularity, so that SM choice is independent of the backup, and vice versa. Unfortunately, the modularity achieved often lacks the expected composition properties one would hope to have.

The most common backup method, depicted in Fig. 1, simply chooses a single static symmetric key k, and (authentically) encrypts the plaintext messages m_i under this key k. The resulting ciphertexts c_i are uploaded to the cloud. In Fig. 1, we call this part "Outsourcing". Then a separate "Compact Secure Storage" component is designed to protect this static secret key k. We use red to indicate a secret value that is stored at rest, making it vulnerable to compromises, to distinguish it from the respective short-lived values used while processing in the "Enc" and "Outsourcing" components. The methods for designing this latter component vary a lot from implementation to implementation.

very common for achieving FS/PCS, and used by the two predominant abstractions of SM: continuous key agreement (CKA) [2] and continuous group key agreement (CGKA) [3].

For example, WhatsApp's solution combines hardware secure modules (HSM) at their premise storing the (high-entropy) encryption key k, and a variant of the OPAQUE password-based key exchange to securely retrieve said key based on a low-entropy password while maintaining an adequate level of security [24]. Password-protected secret sharing (PPSS) [7,35] presents an alternative solution to store the key by secret sharing it among a number of servers. Moreover, other solutions include Updatable oblivious key management [36] and DPaSE [23] with both schemes allowing users to derive strong keys from a password using the help of multiple servers, based on varying trust assumptions.

As we are not going to improve on the "Compact Secure Storage" component in this work, we will assume that it is done "well," although the attacker is allowed to occasionally compromise this component, which is a standard practice in designing and analyzing cryptographic primitives. Unfortunately, when such compromise happens for whatever reason, the attacker gets their hands on the static key k used to encrypt all messages — *both past and present*. Stated differently — irrespective of the advanced FS/PCS security properties achieved by the SM protocol whose (not-explicitly-erased) conversation history is being backed up — the overall protocol offers zero PCS/FS security, as everything is eventually encrypted by the same static key k.[2]

While the loss of FS/PCS is our primary concern, we also mention a few other inefficiencies in doing traditional backups, which our solutions will address.

(a) It is generally neither storage nor bandwidth efficient if every user of a group backs up the same content, instead of doing this once for the entire group.
(b) Somewhat less critically, not reusing the secure messaging application's old ciphertexts $\tilde{c}_1, \ldots, \tilde{c}_n$, which *already have all the desired security properties*, and instead generating new ones appears wasteful, especially if backing up long messages, such a video calls.
(c) One could imagine applications where one would want to only delegate part of the conversation history, without first downloading it from the cloud, and then redacting it appropriately. For example, if law enforcement (LE) got a warrant to observe messages sent by the user so far, it would be nice to give a compact token to LE allowing them to precisely get only prior (but not future) messages. Once again, a fixed encryption key k does not work.[3]

1.2 Our Work: Compact Key Storage

Our high-level idea to address all these deficiencies simultaneously, depicted in Fig. 2, is to create a smarter "Outsourcing" box than simply encrypting the plaintexts with a fixed key. For starters, we could directly reuse the ciphertexts $\tilde{c}_i$ produced by the "Encryption Box" Enc, and focus on outsourcing the symmetric keys s_i used to encrypt these ciphertexts. As a minor gain, this saves time on re-encrypting the plaintext with a separate key and already opens the door for sharing the same backup storage by multiple users who anyway know the same SM secrets $\{\mathrm{s}_i\}$. More importantly, by reducing our problem to that of backing up SM secrets $\{\mathrm{s}_i\}$, we get two advantages.

First, the SM application already solved the hard problem of ensuring that the (dynamically changing) set of people who know current s_n is exactly the same set of people who are entitled to know the plaintext m_n for epoch n. This is once again compatible with our idea to share outsourced storage by multiple people. Second, since these keys $\{\mathrm{s}_i\}$ are computationally indistinguishable from independent uniformly distributed keys, we can try to simultaneously use them as "keys for backing up themselves." Namely, instead of using an *externally generated* key k in the naive Backup solution, we will use the application secrets $\{\mathrm{s}_i\}$ *themselves* to *deterministically* derive and update the compact secret state $\mathrm{st}_1, \mathrm{st}_2, \ldots$ which will be input to the "Compact Secure Storage" box in Fig. 2.

[2] Of course, WhatsApp is well aware of this problem, but their current efforts are all directed at designing harder-to-compromise "Secure Compact Storage" component [24], making it practically harder to get k than to compromise the Double Ratchet keys. Our work will offer a complementary solution.

[3] At least not without doing a special encryption process which will anticipate such a warrant; e.g., using puncturable PRFs [17,38,19].

Compact Key Storage. This leads us to the main technical and conceptual novelty of our work: *Compact Key Storage* (CKS) primitive. CKS dynamically (and deterministically!) transforms the stream of application secrets $s_1, s_2, \ldots$ into a stream of outsourced ciphertexts $c_1, c_2, \ldots$ stored in the cloud, and a compact, dynamically changing secret state st_n, which needs to be protected using whatever "Compact Secure Storage" solution we wish to use.[4] The current state st_n has several attractive features, beyond compactness:

- **Secret Recovery**: despite compactness, st_n plus an appropriate subset of stored ciphertexts $\{c_i\}$ is enough to recover any prior secret s_j, *unless this secret is forgotten* (see "FS" below).
- **PCS**: compromising st_n clearly leaves all future secrets $s_{n+1}, \ldots$ secure, as st_n does not depend on these (computationally) unrelated secrets.
- **FS/Delegation**: users can "forget" some subsets S of the secrets, by transforming their current state st_n into a "lesser" state st'_n. In particular, every secret in S will be "secure" upon corrupting st'_n, even given all previously uploaded ciphertexts $(c_1, \ldots, c_n)$.[5]
- **Deduplication**: since CKS is deterministic, all users knowing st_n and s_{n+1} will compute the same ciphertext c_{n+1} and state st_{n+1}. In particular, only one copy of c_{n+1} should be uploaded to the cloud, by any one of such users.

Our Contributions. We can group our overall contributions into three categories. These are explained in detail in Sections 3, 5, and 4, respectively, but here we mention the highlights:

1. **CKS Primitive.** First, we propose CKS as a novel aspect of outsourced cloud backups in secure messaging, or more generally contexts where FS and PCS are relevant. As such, our work complements the existing literature on backup that predominantly focuses on securely storing and retrieving a small secret state (such as an encryption key) based on a user-memorable password.
2. **Efficient Constructions.** Second, we propose two practically efficient CKS schemes. Our *line scheme* covers the most basic "all-or-nothing" flavor of CKS, where one wishes to compactly store and delegate the entire history of past secrets. In particular, new users enjoy the efficiency and compactness properties of the CKS only after being granted access to the entire history of keys. In contrast, our *interval scheme* is only slightly less efficient but allows for finer-grained access/delegation and selective deletion of past keys.
3. **Novel Definition.** Third, we introduce a novel *security preservation* definitional approach that could be of independent interest, well beyond the setting of CKS. In essence, it applies to security games where the traditional "key indistinguishability" notion is unachievable (which is the case for CKS, as we show in Section 4.2). Instead, our definition requires that the addition of CKS functionality (e.g., the knowledge of uploaded ciphertexts $\{c_i\}$) provably does not hurt the security of the original application (i.e., SM). In fact, we will manage to argue such "security preservation" in quite general terms, albeit in the random oracle model. This allows our definition of CKS to be applicable to many application scenarios (see "Additional CKS Use Cases" below), beyond adding backup to SM. We motivate our notion in Section 4.1.

Additional CKS Use Cases. As mentioned above, the generality of our CKS definition makes it applicable to other application scenarios, beyond secure backup. First, supporting multiple devices has been a long-standing user request for popular E2E-secure messaging applications such as WhatsApp [49]. Recently, WhatsApp has thus re-engineered its application to natively support companion devices, such as browser-based sessions or desktop clients, without requiring an active smartphone connection. According to [49], whenever such a device is linked to the primary phone, the primary device encrypts

[4] As we mentioned, this part does not change, and one still needs to derive a good method to secure this compact state.

[5] This correctly models forward secrecy in the backup application scenario, where our goal is to keep the entire conversation history, unless *explicitly erased*.

a bundle of recent chat messages and transfers them to the newly linked device.[6] Instead, leveraging CKS one could have the primary device simply send its current compact state (per group) to the new device.[7]

Our CKS notion furthermore supports fine-grained (say, given by a date range) delegation of parts of the conversation to another party. This has several use cases from auditing and compliance to enabling granting access to (select) past conversations to new group members. (For instance, when a party replies to a message, one may show that old message also to parties having joined in the meantime. There also exists a number of applications such as Slack or Keybase where new group members get access to the entire chat history by default.)

Finally, while conceived in the context of secure messaging, CKS composes with any application that leverages some ordered sequence of shared keys. In particular, this covers all applications using the continuous key agreement (GKA) [2] and continuous group key agreement (CGKA) [3] abstractions. For instance, Zoom's E2EE video meetings [26] use a form of CGKA that allows to rotate the keys used to encrypt the meeting streams as participants join or leave the meeting. One could, for instance, leverage CKS to build more efficient cloud recordings of E2E encrypted video calls where the server directly records the ciphertexts, while parties use the delegation process of the compact key storage to share the recording with outsiders.

Practical Aspects. Our schemes are based on symmetric encryption and hash functions — some of the most efficient cryptographic primitives. Amortized, each party performs a constant number of such operations per message. The scheme can be run in the background and is not latency-critical. In terms of cloud storage, we expect the scheme to become more efficient for groups with more than a handful of participants when compared to the existing per-user backup with static keys (while providing superior functionality even for smaller groups).

When compared to the traditional backup process with static keys, the on-device backup process becomes slightly more involved. First, the required secret state evolves over time (this is inherent to obtaining FS/PCS). At the same time, the secret state must still be recoverable if the user loses all their devices, meaning it should preferably be protected by a static human-memorable secret. As discussed earlier, efficient solutions for using human-memorable secrets to protect a static secret typically involve hardware secure modules (HSM) [24] or password-protected secret sharing (PPSS) [7,35]. We consider extending those approaches to support changing secrets (securely erasing the old ones) to be an orthogonal issue to CKS. Note also that for our interval scheme, the secret state (slowly) grows over time. If an issue, one can employ an additional layer of encryption to store this growing CKS state in the cloud under a freshly sampled fixed-size key, and then protect that latter key instead. Given the compact nature of the CKS state, we do not anticipate issues uploading such encrypted CKS state in regular intervals, such as once a week.

1.3 Scheme Overview

Let us give an overview of our line scheme, which covers the most basic "all-or-nothing" flavor of CKS. Consider some E2E-secure application that wishes to securely back up a sequence of secrets $(\mathrm{s}_1, \ldots, \mathrm{s}_n)$. For simplicity, we assume in the following that parties learn those secrets in order; we refer to Section 5.1 on how to handle the general case. On a very high level, our CKS scheme will iteratively aggregate a (secret) state $\mathrm{st}_{\mathsf{e}-1}$ and a secret s_{e} into a new compact state st_{e} and a ciphertext c_{e} to be outsourced. The state st_{e} together with c_{e} then allows the party to later recover $\mathrm{st}_{\mathsf{e}-1}$ and s_{e}. Hence, the party can recover all secrets $(\mathrm{s}_1, \ldots, \mathrm{s}_{\mathsf{e}})$ from st_{e} and $(\mathrm{c}_1, \ldots, \mathrm{c}_{\mathsf{e}})$. See Fig. 3 for a schematic depiction.

[6] To the best of our understanding, the reason for WhatsApp to implement multi-device support in this manner is the opt-in nature of backups.

[7] The trust model here slightly differs from sharing with another party, as the user's primary device can be assumed to be honest. Also, fine-grained delegation is typically irrelevant for adding additional devices to the same user account.

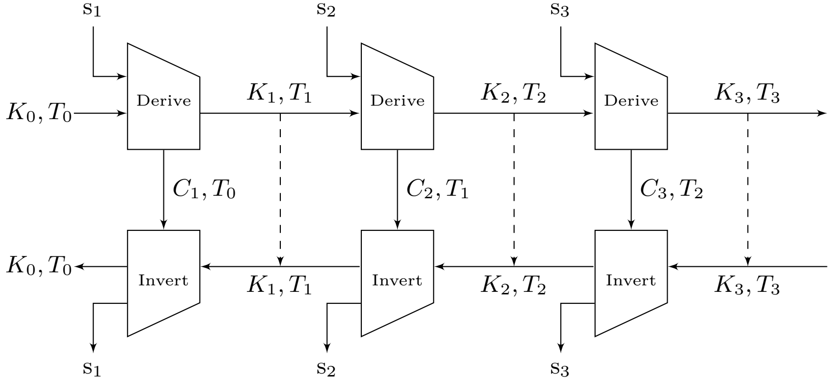

Fig. 3: A schematic representation of line CKS scheme. The top half shows the parties outsourcing the keys, while the lower half shows the key-recovery process.

To achieve deduplication across parties, we need this process to be deterministic and not assume any additional shared secrets across parties beyond $(s_1, \dots, s_n)$. In other words, we want that each party that knows the same sequence of secrets to end up with the identical sequence of ciphertext $(c_1, \dots, c_n)$.[8] To achieve this, our construction makes use of Convergent Encryption (CE) as introduced by Douceur et al. [27] for de-duplicated outsourced storage. CE uses a one-time secure *deterministic* symmetric encryption scheme $\mathrm{SE} := (\mathrm{SE.Enc}, \mathrm{SE.Dec})$ and a hash function H. Simply speaking, CE first deterministically derives a key K by hashing the to-be encrypted message itself, and then encrypts it using SE and the derived key. Therefore, the resulting ciphertext C is deterministically derived from the message only, with security intuitively holding in the ROM only for high-entropy messages. In addition to the ciphertext, CE also computes a tag T authenticating the ciphertext.

In our CKS protocol, a state $\mathrm{st}_\mathsf{e} = (\mathsf{e}, K_\mathsf{e}, T_\mathsf{e})$ stores the latest epoch number, CE key, and CE tag. A ciphertext $c_\mathsf{e} = (\mathsf{e}, C_\mathsf{e}, T_{\mathsf{e}-1})$ sent to the server contains the CE ciphertext and the prior tag.[9] We call the aforementioned aggregation process "Derive" and implement it based on CE:

Derive$\big(s_\mathsf{e},\ \mathrm{st}_{\mathsf{e}-1} = (\mathsf{e}-1, K_{\mathsf{e}-1}, T_{\mathsf{e}-1})\big)$:
1. $K_\mathsf{e} \leftarrow \mathrm{H}(s_\mathsf{e} \| K_{\mathsf{e}-1})$ computes the new key based on the new secret s_e and the previous key;
2. $C_\mathsf{e} \leftarrow \mathrm{SE.Enc}(K_\mathsf{e}, (s_\mathsf{e} \| K_{\mathsf{e}-1}))$ encrypts the new secret and the old key under the (message-derived) key;
3. $T_\mathsf{e} \leftarrow \mathrm{H}(C_\mathsf{e} \| T_{\mathsf{e}-1})$ generates an authentication tag;
4. Sends $c_\mathsf{e} \leftarrow (\mathsf{e}, C_\mathsf{e}, T_{\mathsf{e}-1})$ to the server and stores $\mathrm{st}_\mathsf{e} \leftarrow (\mathsf{e}, K_\mathsf{e}, T_\mathsf{e})$.

The initial key K_0 and tag T_0 are set to some public constant. The corresponding "Invert" procedure then recovers the secret and prior state by undoing those operations:

Invert$\big(\mathrm{st}_\mathsf{e} = (\mathsf{e}, K_\mathsf{e}, T_\mathsf{e}),\ c_\mathsf{e} = (\mathsf{e}, C_\mathsf{e}, T_{\mathsf{e}-1})\big)$:
1. Recomputes $T'_\mathsf{e} \leftarrow \mathrm{H}(C_\mathsf{e} \| T_{\mathsf{e}-1})$;
2. Checks that $T'_\mathsf{e} \stackrel{?}{=} T_\mathsf{e}$;
3. Decrypts $(s_\mathsf{e} \| K_{\mathsf{e}-1}) \leftarrow \mathrm{SE.Dec}(K_\mathsf{e}, C_\mathsf{e})$;
4. Stores $\mathrm{st}_{\mathsf{e}-1} \leftarrow (\mathsf{e}-1, K_{\mathsf{e}-1}, T_{\mathsf{e}-1})$ and outputs s_e.

Intuitively, security follows from C_e not leaking information about s_e and $K_{\mathsf{e}-1}$ as long as either of them is unpredictable (in the ROM). The precise security statement is, however, a bit subtle and, in

[8] Having identical ciphertexts allows us to efficiently thwart insider attacks in which a party knowing $(s_1, \dots, s_n)$ attempts to disrupt another party's ability to recover secrets or even have them recover wrong secrets.

[9] Including the prior allows the server to lookup the entire sequence of ciphertexts $(c_1, \dots, c_\mathsf{e})$ non-interactively, and the client to verify integrity without decryption.

particular, does not follow from the message-locked encryption (MLE) abstraction of CE by Bellare et al. [10].

The interval scheme works similarly by organizing the secrets $(\mathrm{s}_1, \ldots, \mathrm{s}_n)$ as leaves of a binary tree. The "Derive" process is then used to aggregate two children into their parent node, thus aggregating two secrets $(\mathrm{s}_i, \mathrm{s}_{i+1})$ at the first level and two CE keys on higher levels. Each intermediate node thus has an associated CE key, tag, and ciphertext.

1.4 Related Work

Outsourced storage with PCS and FS, as well as secure deduplication mechanisms, have been studied extensively. To the best of our knowledge, they have however been treated mostly separately, while our work combines both aspects.

Updatable encryption. A line of research [16,29,40,39,18,15] tackling the former issue — in particular PCS — is updatable encryption (UE), which allows a party to periodically rotate encryption keys, and then issue a special "update token" Δ, using which another party can update ciphertexts encrypted under old key k to those decryptable using a new key k'. In particular, UE can be applied to the naive Backup scheme from Fig. 1 to rotate the static outsourcing key k. Unfortunately, doing so has several disadvantages when compared to the CKS solution from Fig. 2, in addition to not addressing the deduplication issue.

First, achieving PCS forces the cloud server to re-encrypt *all* prior ciphertexts after receiving the update token Δ. Second, UE solutions either do not address the FS issue at all (e.g., bi-directional ciphertext-independent schemes), or address it in a computationally expensive way (e.g., uni-directional ciphertext-independent and ciphertext-dependent schemes). For example, in the case of ciphertext-dependent schemes, the user would need to download all the ciphertexts they want to "keep," before issuing a compact token to the server, which then will update all such ciphertexts. A notable exception is the ciphertext-independent scheme by [47], which achieves FS using heavy public-key machinery, while still forcing the server to manually update all the ciphertexts the user wants to keep.

Secure deduplication. The second issue is tackled by secure deduplication schemes. Message Locked Encryption (MLE) [10,9] and Multi-Key Revealing Encryption [41] allow servers to deduplicate encrypted storage of the same file by different users. In particular, the simplest MLE scheme, called convergent encryption [27], symmetrically encrypts the message using as a key the hash of the message itself, so that encrypting the same message will result in the same ciphertext. More recently, secure deduplication has been studied by Best et al. [12], who propose a protocol satisfying Universal Composability (UC) security.

We note, however, that those schemes do not address the issue of incrementality growing the backup as new messages arrive. More concretely, when applied to the naive outsourcing solution, parties would either have to encrypt the new data separately, growing the compact secure storage linearly, or re-encrypt the entire communication history each time. Nevertheless, our CKS schemes use convergent encryption [27] mentioned above, but at a lower level. This allows us to fix the incrementality issue, while also addressing the FS/PCS/delegation concerns (which are not addressed by any existing secure deduplication primitives).

Additional related work. Many additional concerns about outsourced storage have been considered. These concerns are generally orthogonal to CKS but sometimes can be added to any CKS scheme in a black-box manner. Additionally, none of them addresses the deduplication aspect of CKS, as all these solutions require an independently sampled secret key.

The first line of work is focused on improved confidentiality. Oblivious RAM (ORAM) [32,45,33,48] [44,5] allows a single client to outsource a large amount of storage to an untrusted server while allowing for efficient retrieval and update of individual memory chunks, hiding not only the content of such chunks but also the access patterns. Forward Secure Encrypted RAM (FS-eRAM) [13,25] does not

hide access patterns, but ensures that even if both the client and server are compromised, content that had been erased or overwritten cannot be recovered.

Another direction is protecting data against tampering and deletion. Memory Checkers [14,43,21] focus on ensuring the integrity of a portion of the outsourced storage that is getting accessed, without providing any confidentiality. Proofs of Retrievability (also Proofs of Storage) [46,37,6,28] allow a server storing a large amount of data on behalf of a client to prove that such data hasn't been erased, with far smaller communication than the size of the data being stored. Similarly, Filecoin [31] leverages Proofs of Replication [4,22,30] to offer distributed storage as a service (from a set of untrusted servers), but again it does not allow multiple clients to easily cooperate or offer interfaces for efficient delegation and independent updates.

1.5 Outline

In Section 3 we introduce the CKS notion. In Section 4 we then define CKS security. We highlight the challenge of defining security and prove that the strongest and most natural notion of key indistinguishability is, unfortunately, impossible in Sections 4.1 and 4.2, respectively. In Section 4.3 we introduce our novel approach which we call preservation-security. Furthermore, we present a strong integrity notion protecting users from restoring erroneous data even if both the cloud server and other users of the system behave maliciously. In Section 5 we present two CKS schemes. The first one is a simple "all-or-nothing" CKS scheme in which users either know all or none of the keys. The second is an improved, albeit slightly less efficient scheme, that allows for fine-grained delegation and erasure (for FS).

2 Preliminaries

In this work, we 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.

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 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. We present some additional preliminaries in Appendix A.

3 Compact Key Storage

3.1 High-level CKS Overview

Recall from Fig. 2 that the goal of CKS is to create a smarter "Outsourcing" box by reusing the ciphertexts $\tilde{c}_i$ produced by the "Encryption Box" Enc, and focus on outsourcing the symmetric keys

s_i used to encrypt these ciphertexts instead. For each epoch n, given current compact state st_{n-1} and a new secret s_n, CKS thus produces a small local state st_n (using a *non-interactive* Append procedure), and (potentially) uploads some ciphertext(s) c_n to the server (using the corresponding Upload procedure).

Accessing keys. Whenever a user U wants to retrieve one or more of the keys s_i that it knew at some point, they can run the Retrieve procedure with the server to download the ciphertexts "necessary" to retrieve these keys using their current compact state st_n. For this to succeed, obviously the necessary information must have been uploaded to the server. More concretely, we require that at least one user V who knew "as much as U" at epoch i (which includes U itself, of course) ran the Upload procedure in that epoch. We will elaborate on this when discussing correctness.

Delegating access. When a new user V joins the system (e.g., becomes a member of the secure messaging group), any existing user U can share her current CKS state st_n with V to immediately give V access to everything that U knows. Crucially, we want this delegation operation to be more efficient than the trivial solution of having the user U recover all keys locally, and (privately) sending them to the other party V. Both the granting operation by U and the accepting operation by V might use the help of the server S. Concretely, when U runs the Grant operation with S, it will obtain some compact message msg that U will securely transmit to V.[10] V will then use msg to run the corresponding Accept procedure with S to appropriately update its secret state st_n.

Erasing secrets. A special case of delegation could be self-delegation. More generally, a user U might wish to explicitly "forget" some of the secrets that it no longer needs.

Supported subsets. We were quite general in terms of functionality so far, potentially allowing the user to outsource, retrieve, and delegate arbitrary subsets of secrets. Unfortunately, such generality cannot be supported efficiently. As a result, our notion will be parameterized by three respective predicates — one for retrieval, delegation, and erasure — which will govern for which subsets of keys the CKS must be able to efficiently operate the respective functionality.

3.2 Syntax Definition

We now formally define the Compact Key Storage notion. To this end, we first introduce a notion formalizing which subsets of keys users can efficiently delegate, which then becomes a parameter of the CKS notion.

Definition 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* $\mathrm{know} \subseteq \mathbb{N}$ *of epochs denoting the respective keys known to a party,* $\mathcal{G}(\mathrm{know}, \mathrm{share})$ *indicates whether they can delegate* $\mathrm{share} \subseteq \mathbb{N}$*. Analogously, a* retrieval family $\mathcal{R}$ *and an* erasure family $\mathcal{E}$ *indicate whether the party can recover* $\mathrm{share} \subseteq \mathbb{N}$ *or erase* $\mathrm{share} \subseteq \mathbb{N}$*, respectively.*

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

[10] Any protocol that supports dynamic groups needs some setup assumptions to bootstrap security. This could be a traditional PKI, security codes, or a key-transparency system. We leave the choice to the outside application and simply assume that there is a way for the message to be securely communicated.

Key Management:

- *The non-interactive append algorithm takes the current state* st*, an epoch* $\mathtt{e}$*, a secret* s*, and flag* upload*. The invocation*
$$\big(\mathrm{st}', \mathrm{st}_{\mathrm{up}}\big) \leftarrow \mathrm{U.Append}(\mathrm{st}, \mathtt{e}, \mathrm{s}, \mathrm{upload}),$$
produces an updated state st' *and, if* upload $=$ `true`*, an upload state* $\mathrm{st}_{\mathrm{up}}$*. (If* upload $=$ `false`*, then* $\mathrm{st}_{\mathrm{up}} = \bot$*.)*
- *The interactive upload algorithm takes the upload state and after the interaction*
$$\big(\bot; \mathrm{st}_{\mathrm{S}}'\big) \leftarrow \big\langle \mathrm{U.Upload}(\mathrm{st}_{\mathrm{up}}) \leftrightarrow \mathrm{S.Upload}(\mathrm{st}_{\mathrm{S}})\big\rangle,$$
the server outputs an updated state $\mathrm{st}_{\mathrm{S}}'$*.*
- *The interactive erase algorithm takes the current state* st *and a set of epochs* share $\subseteq \mathbb{N}$*. After the following interaction*
$$\big(\mathrm{st}'; \bot\big) \leftarrow \big\langle \mathrm{U.Erase}(\mathrm{st}, \mathrm{share}) \leftrightarrow \mathrm{S.Erase}(\mathrm{st}_{\mathrm{S}})\big\rangle,$$
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* share $\subseteq \mathbb{N}$ *of keys to be shared with another user* U_2*. After the interaction*
$$\big(\mathrm{msg}; \bot\big) \leftarrow \big\langle \mathrm{U}_1.\mathrm{Grant}(\mathrm{st}_1, \mathrm{share}) \leftrightarrow \mathrm{S.Grant}(\mathrm{st}_{\mathrm{S}})\big\rangle$$
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*
$$\big(\mathrm{st}_2', \mathrm{st}_{\mathrm{up}}; \bot\big) \leftarrow \big\langle \mathrm{U}_2.\mathrm{Accept}(\mathrm{st}_2, \mathrm{share}, \mathrm{msg}, \mathrm{upload}) \leftrightarrow \mathrm{S.Accept}(\mathrm{st}_{\mathrm{S}})\big\rangle$$
the user outputs an updated state st_2'*, as well as (if* upload $=$ `true`*) a state for the* Upload *algorithm.*

Retrieval:

- *The interactive key-retrieval algorithm restores the secrets for epochs* share $\subseteq \mathbb{N}$ *with the interaction*
$$\big(\mathrm{secrets}; \bot\big) \leftarrow \big\langle \mathrm{U.Retrieve}(\mathrm{st}, \mathrm{share}) \leftrightarrow \mathrm{S.Retrieve}(\mathrm{st}_{\mathrm{S}})\big\rangle$$
ending with the user outputting a function secrets: share $\rightarrow$ s*.*

Efficiency requirements. In the following, assume secrets are appended in (roughly) consecutive order. For a CKS scheme to be considered efficient, we require that in this case all operations operate in sublinear — ideally logarithmic — time in the number of epochs n. 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 $\mathcal{I} = \mathcal{I}_1 \cup \cdots \cup \mathcal{I}_k$ and obtain 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. Observe that in the above definition, the server state is only modified in Upload. For simplicity, we do not model that Erase might enable to server to discard state once no party needs the certain information anymore; an actual implementation might of course implement such optimizations.

3.3 Correctness

Honest parties interacting with an honest server must be able to retrieve their secrets, as long as uploading to the server is properly coordinated. This coordination can be non-trivial.[11] In particular, it may be insufficient if only the first user to learn a new secret runs the Upload procedure. Instead, we may want to allow users to learn different subsets of secrets, learn secrets in different order, or learn inconsistent secrets altogether. For instance, consider a setting with $n+1$ users and n epochs. For $j \leq n$, user U_j learns s_j (and nothing else). Afterwards, U_{n+1} learns all secrets $\mathrm{s}_1, \ldots, \mathrm{s}_n$. For efficient protocols, we cannot expect this user to maintain a compact state that allows them to recover all secrets without having uploaded information themselves.[12]

We thus mandate that uploading is required unless another party knows a superset of secrets for the same set of epochs. For example, if a user U already knows secrets 1 to n and now learns secret $n+1$, then they are not required to upload (but still can!) in case another user U' already knows secrets 1 to $2n$. If, on the other hand, at this point, U' knows secrets 1 to n and U'' knows $n+1$ to $2n$, then U still must run Upload. We believe this to be a good trade-off between usability (e.g., not requiring parties to learn secrets in the same order to be able to "contribute") and efficiency of potential schemes.

To enhance composability, we moreover demand that correctness holds even if the keys might be adversarially chosen, and all states are assumed to be known to the adversary. In the case of a user accepting a malicious grant request, no correctness guarantees are given for those particular keys but the correctness of other epochs must not be affected. More concretely, the game tracks the difference between epochs for which they know a certain secret (as tracked by Secret) and those for which they believe to know a key (as tracked by Known) with the two diverging in case of malicious delegations. The game then requires that retrieving and delegating keys works as long as all involved keys are honest, and that erasure works as long as at least one key is honest. Furthermore, the adversary can emulate malicious insiders interacting with the server. For simplicity, we assume the adversary to not interleave those oracle calls, as an honest server could always serialize requests. Finally, note how $(\mathcal{G}, \mathcal{R}, \mathcal{E})$ affects the correctness notion. For instance, the *Grant* oracle can only be invoked for a user U and share such that the user's knowledge satisfies $\mathcal{G}$ for this share.

Definition 3. *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. 4 is negligible in* κ.

4 CKS Security

In this section, we define the security of a CKS scheme, which consists of two properties: confidentiality and integrity.

4.1 Definitional Challenges

Recall that CKS is not so much a cryptographic application in its own right, but intended to augment existing applications with backup and delegation functionality. As such, the fundamental goal must be to define CKS security in such a way that a (presumably) secure CKS scheme does not undermine the security of the higher-level application. Moreover, one would wish for this to be modular such that *any* secure CKS scheme can be used to augment *any* (suitable) application without having to worry about interdependence. Typically, we expect this process to look as follows:

[11] The coordination is left to the application integrating CKS. One possibility is each client communicating with the server to see whether uploading is required. An alternative is the server issuing an attestation to the client that did the upload, which can then be shared with the other clients.

[12] It seems that such a strong notion would imply some form of homomorphic encryption where the server is able to combine the information from the other users.

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

```
Main
  win ← false, n ← 0
  St[·], Secret[·,·], GrantInfo[·] ← ⊥
  Known[·,·] ← 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 Secret[u, e] = ⊥
  Secret[u, e] ← s; Known[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(Known[u, ·], share)
  (st′; ⊥) ← ⟨U.Erase(St[u], share) ↔ S.Erase(st_S)⟩
  if st′ = ERROR then
    if ∀e ∈ share : Secret[u, e] ≠ ⊥ then win ← true
  else
    St[u] ← st′
    for e ∈ share do
      Known[u, e] ← false; Secret[u, e] ← ⊥
  return st′

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

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

Helper Subsumed
Input: u ∈ [n]
  return ∃u′ ≠ u ∈ [n], ∀e : Secret[u, e] ∈ {⊥, Secret[u′, e]}
```

Fig. 4: The Compact Key Storage correctness notion. We assume the adversary to not interleave calls of the adversarial oracles.

1. Prove the higher-level application (which uses a sequence of secrets) secure, without concerning CKS;
2. Prove a CKS scheme secure with respect to its standalone security definition;
3. Deduce that the combined scheme, where one outsources the sequence of secrets using CKS, is still as secure as the original application.

In particular, we would like that the last step can be done generically, reducing any attack on the combined scheme to one on either the original scheme or the CKS security. Therefore, one can employ any CKS scheme for any given application, and prove the security of each CKS scheme in isolation.

The natural candidate for a CKS security notion satisfying the above constraints would be pseudorandomness: All the real secrets should remain pseudorandom even when given the CKS ciphertexts, i.e., the uploaded backup. Indeed, most applications that use secrets assume in their security games that those are chosen uniformly at random (or at least can be modularized into a part

establishing pseudorandom secrets and a part assuming random secrets). Unfortunately, we will show in Section 4.2 such a notion to be impossible, i.e., not satisfiable by any CKS scheme. We furthermore argue in Section 4.3 that this is not simply a definitional shortcoming but that there are indeed applications which when composed with our proposed CKS scheme — and most likely any efficient CKS scheme — would be rendered insecure.

Slightly less ambitious, we can therefore ask for the combination with CKS to be secure for *a wide range of applications*. How should this be reflected in the CKS security definition, and how can the class of applications that can be combined with CKS be characterized? One approach would be something along the line of saying that the secrets remain unpredictable even given the CKS ciphertexts. Indeed, such a definition seems possible and avoids the impossibility result of Section 4.2. However, this severely limits the class of potential applications that can be CKS-augmented, as many applications require random and not merely unpredictable keys for their security. While, at least in the ROM, most applications could be adjusted to work with unpredictable keys, a goal of ours is to be able to combine CKS with *legacy applications* such as Signal or MLS that have not been designed with CKS in mind.

Traditionally defining a weaker security notion for an abstract class of "good" applications is done using two-stage games. Prominent examples are MLE or the notion of Universal Computational Extractors (UCE) [11]. For instance, when applied to CKS one could formalize that a first adversary generates some leakage (modeling the usage of the secrets by an underlying application such as SM), while a second adversary then tries to break CKS. Unfortunately, such two-stage games tend not to survive contact with reality. For instance, the sketched definition for CKS would require the underlying application (e.g., SM) to be completely independent of the CKS. This assumption is trivially broken the moment any malicious insider sends a CKS ciphertext over the secure messaging scheme. Generalizing such multi-stage games to rule out impossibilities (i.e., impose enough restrictions) while still capturing a wide variety of applications has proven to be challenging for various notions such as UCE [20,11].

A novel approach. Instead of using a multi-stage game, in the remainder of the section, we propose a novel approach to defining CKS security, that (a) is (almost) independent of the concrete application; and (b) appears to compose better than multi-stage games. In a nutshell, we turn the definition on its head, and instead of formalizing CKS security in isolation, we define a very broad (and easy to understand) class of "CKS-compatible games" G modeling the higher-level application (e.g., SM) which uses the keys stored by the CKS. In Appendix B we show that this class encompasses a wide variety of protocols, including Signal. We then require that CKS does not undermine the security of *any* such CKS-compatible game G. We call this *preservation security*, as it aims to prove that CKS preserves the security of an (already analyzed) scheme.

Our approach aims to bring the best of both worlds and achieve a notion that can be met by (practical) CKS schemes while general enough to compose with a wide variety of applications. We stress that our approach still constitutes a proper modularization, extricating us from proving the security of each concrete application with a concrete CKS scheme as a monolith. We hope that this approach could also be useful for other primitives such as MLE [10].

Modeling choices. While we strive to keep our novel security definition as generic as possible, some of its aspects do reflect specific assumptions about the schemes. Let us briefly discuss those:

- *Deterministic schemes:* Our security definition deliberately allows for deterministic CKS schemes. Recall that CKS should allow a group of parties to jointly outsource a shared sequence of secrets, without additional interaction, coordination, or shared setup. This seems inherently linked to various deduplication notions whose practical schemes share the trait of being built around determinism. Examples are MLE or proof-of-ownership [34].
- *ROM:* Our security definition will leverage the random oracle model. We believe that most practical CKS schemes will anyway rely on the ROM.

4.2 Impossibility of Key-Indistinguishability

For simplicity, consider the special case of "all-or-nothing" CKS. Recall that in this case the functionality of the scheme simply requires that from the current state st_i and previously uploaded ciphertexts $(\mathrm{c}_1, \ldots, \mathrm{c}_i)$, all the secrets $(\mathrm{s}_1, \ldots, \mathrm{s}_i)$ should be recoverable. Now, we would like to say that the knowledge of ciphertexts should not jeopardize the security of the outside application that uses $(\mathrm{s}_1, \ldots, \mathrm{s}_i)$.

The strongest and most natural formalization of such security would require that all the real secrets remain pseudorandom conditioned on $(\mathrm{st}_0, \mathrm{c}_1, \ldots, \mathrm{c}_i)$, where st_0 is the initial (public) state. Unfortunately, a moment of reflection demonstrates that this is incompatible with correctness — requiring that, from the current state st_i and previously uploaded ciphertexts $(\mathrm{c}_1, \ldots, \mathrm{c}_i)$, all the seeds $(\mathrm{s}_1, \ldots, \mathrm{s}_i)$ should be recoverable — and compactness saying that correctness should hold even if the ciphertexts were uploaded by *different* users.

This impossibility holds true even for a weaker notion with an honest-but-curious server. To this end, consider the following simple key-indistinguishability notion: One user, say, Alice, appends n keys to the CKS. Afterward, an adversary is given real-or-random keys as well as the server's most recent state st_S and tries to distinguish the two cases.

Lemma 1. *For every correct CKS scheme that is non-trivial — i.e., for which a party's local state grows slower than the key material recoverable by that party — there exists a PPT adversary $\mathcal{A}$ and a polynomial n, such that*

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

is not bounded by $\frac{1}{2} + \mathsf{negl}(\kappa)$.

Proof (Sketch). Choose n such that the compactness of the CKS kicks in: $|\mathrm{st}_n| \ll |\mathrm{s}_1| + \ldots + |\mathrm{s}_n|$, where st_i refers to the state after the i-th append operation in the above experiment. In the following, let's call the user appending the keys to the CKS in the above experiment Alice. Now, consider the following adversary $\mathcal{A}$: The adversary emulates a second party, say, Bob in their head. Bob appends the keys $\mathrm{s}_1^b, \ldots, \mathrm{s}_n^b$ to the CKS by internally running Append. Finally, $\mathcal{A}$ retrieves all the keys on Bob's behalf by internally emulating the interactive Retrieve algorithm using the state $\mathrm{st_S}$ input to the adversary. It outputs 0 iff all keys match the one input to the adversary.

Observe that by correctness $\mathcal{A}$ will output 0 if $b = 0$, i.e. if Bob uploads the same keys as Alice. On the other hand, if $b = 1$ then the server's state $\mathrm{st_S}$ at the end of Alice's uploads is independent of the keys $(\mathrm{s}_1^1, \ldots, \mathrm{s}_n^1)$ afterward output by the Challenge oracle. Observe that $\mathcal{A}$ uses this same unmodified server state throughout the rest of the execution (recall that for correctness only one party needs to upload) and in particular for the final key retrievals. Meanwhile Bob's state st'_n can depend on the keys input to the adversary — but is compact as well, i.e., $|\mathrm{st}'_n| \ll |\mathrm{s}_1| + \ldots + |\mathrm{s}_n|$. Hence, the probability of the interaction of Bob and the server successfully recovering $(\mathrm{s}_1^1, \ldots, \mathrm{s}_n^1)$ is negligible based on a basic information-theoretic observation. □

4.3 Preservation Security

In this section, we formally introduce our CKS security notion.

CKS-compatible games. First, we define the type of games for which preservation security will be applicable, which we call *CKS compatible*. Obviously, such a game (and hence its corresponding primitive) must define keys one wishes for CKS to outsource. We stress that this does not necessarily mean that the adversary gets to see those keys, but simply that syntactically the game defines such a sequence. For instance, for any secure messaging application that can be modularized as employing a continuous key agreement (CKA) [2] or continuous group key agreement (CGKA) [3] scheme, the respective security game hopefully turns out to be CKS-compatible with the C(G)KA instance run as part of the scheme within the challenger defining the sequence of keys.

To cope with corruptions of a party's CKS state, a CKS-compatible game must moreover offer an *exposure oracle* leaking the keys: By correctness the CKS state allows the adversary to restore all its stored keys, implying they can no longer be secure. It is up to the game to deem certain unavoidable attacks (e.g., where the key used for a challenge of a committing-encryption scheme would be exposed) caused by such an exposure to be "trivial" by adjusting its winning condition accordingly (e.g., deeming the adversary to lose the game if they exposed a key used in a challenge).

CKS schemes must support effective deduplication and strong integrity properties against malicious insiders. We anticipate that this implies efficient schemes to leverage determinism. (For instance, for MLE, no randomized scheme with strong tag consistency is known [10].) As a result, we stipulate that CKS must only be used for applications that can tolerate a *testing oracle* for the keys without security breaking down.

Definition 4. *A security game* $\mathsf{G} = (\mathcal{C}, \alpha)$ *is characterized by a challenger* $\mathcal{C}$*, which upon interaction with an adversary* $\mathcal{A}$ *outputs a bit* b*, and the advantage function* $\alpha \colon \mathbb{N} \to [0, 1]$ *such that the advantage of* $\mathcal{A}$ *winning* G *is characterized by*

$$\mathsf{Adv}_{\mathsf{G}}(\mathcal{A}) = |\Pr[\mathcal{A}(1^\kappa) \leftrightarrow \mathcal{C}(1^\kappa) \Rightarrow 1] - \alpha(\kappa)|,$$

where the interaction's output bit is w.l.o.g. decided by the challenger $\mathcal{C}$*.*

Definition 5. *A game* $\mathsf{G} = (\mathcal{C}, \alpha)$ *is said to be* CKS compatible *if* $\mathcal{C}$ *provides the following private oracle (not accessible to the adversary)*

- *A* Keys *oracle that upon input an epoch* $e \in \mathbb{N}$ *outputs a key* $k \in \{0, 1\}^\kappa$ *or an error* $\perp$*, such that once a value* k *has been set it is persistent (i.e., subsequent queries return the same* k*).*

and the following public oracles to the adversary

- *A* Test *oracle that upon input an epoch* e *and a key* k' *outputs whether* $k = k'$*, i.e., whether* k' *matches the return value by the* Keys *oracle.*
- *An* Expose *oracles that upon input an epoch* e *returns the key from* $\mathsf{Keys}(e)$*.*

Keys are to be of fixed length and, for simplicity, we assume the key space to be $\{0, 1\}^\kappa$ *in the following.*

Examples of CKS compatibility. CKS compatibility is mainly a structural requirement — the application must make use of an ordered sequence of secrets — except for the key-testing oracle. Most cryptographic primitives and schemes naturally tolerate the presence of such a testing oracle, as any attacker who could guess secrets with non-negligible probability could anyway break the primitive/application. The exception to this rule appears to be non-committing primitives. In Appendix B we indeed show the CKS compatibility of a number of primitives, and also briefly sketch the CKS compatibility of the Signal protocol, one of our main motivating examples for CKS.

In the following, we consider two simple examples. First, we establish the CKS compatibility of one-time authenticated encryption — the argument that will be at the core of Signal's CKS compatibility. Second, we use the One Time Pad as a rare example of an application that cannot be securely augmented with CKS. For simplicity, we consider a single epoch (i.e., a single secret) only and only allow for a single call to the testing oracle. Both simplifications are without loss of generality.

Positive example: one-time authenticated encryption. Consider (symmetric) authenticated encryption with one-time security, which formalizes the security of an AE scheme where at most one message can be encrypted. Intuitively, we argue that any adversary who can guess the key could instead break authenticity. As a result, the testing oracle can be replaced by a trivial one that always returns 0.

Lemma 2. *Let $\mathsf{G}_{\mathsf{OTAE}}$ denote the one-time AE security game: if $b = 0$ then the adversary can ask for the encryption of a single challenge message m^* and receive c^*. Afterward, they can make arbitrarily many decryption queries for $c' \neq c^*$. If $b = 1$, then the game returns c^* to be a uniform random ciphertext instead, and the decryption oracle always returns $\perp$. Let $\mathsf{G}^+_{\mathsf{OTAE}}$ denote the same game with an additional testing oracle for the key k that can be queried once. For every PPT adversary $\mathcal{A}$ that can win $\mathsf{G}^+_{\mathsf{OTAE}}$ with non-negligible probability, there exists a PPT adversary $\mathcal{B}$ that wins $\mathsf{G}_{\mathsf{OTAE}}$ with non-negligible probability.*

Proof. Consider a hybrid game $\mathsf{H}_{\mathsf{OTAE}}$ that augments $\mathsf{G}_{\mathsf{OTAE}}$ by a trivial testing oracle that always returns 0. As this oracle can be emulated, winning this game is just as hard as winning the standard PRF security game $\mathsf{G}_{\mathsf{OTAE}}$. To see that any PPT adversary $\mathcal{A}$ has the same winning probability for the hybrid as the game $\mathsf{G}^+_{\mathsf{OTAE}}$ with a proper testing oracle, we use the following reduction to $\mathsf{G}_{\mathsf{OTAE}}$:

- The adversary $\mathcal{B}$ runs $\mathcal{A}$ internally, forwarding the challenge message and response, as well as any decryption queries.
- Once $\mathcal{A}$ calls the testing oracle on k', $\mathcal{B}$ randomly chooses $\tilde{m}$ different to m' and encrypts it under k' to obtain $\tilde{c}$.
- It then submits $\tilde{c}$ to the decryption oracle and returns $b' = 0$ if this decrypts to $\tilde{m}$. Otherwise, it samples $b' = 1$ uniformly at random.

Assume $\mathcal{A}$ distinguishes $\mathsf{G}^+_{\mathsf{OTA}}$ and $\mathsf{H}_{\mathsf{OTA}}$ with probability β. Since the games are equivalent-until-bad, this means $\mathcal{A}$ queries the testing oracle with $k' = k$ with probability (at least) β. Let γ denote the probability of the check in the reduction succeeding. If $b = 1$, then γ is 0 and, therefore, $\mathcal{B}$ guesses $b' = 1$ with probability 1/2. If $b = 0$, then observe that $\tilde{c} \neq c^*$ with overwhelming probability by the correctness of the scheme. Thus, we know that $\gamma \geq \beta$ (the ciphertext could still decrypt even if the key differs e.g. in one bit). The conditional probability of $\mathcal{B}$ correctly guessing $b' = 0$ is therefore $\gamma/2 + 1/2 \geq \beta/2 + 1/2$, resulting in a non-negligible overall advantage. □

Negative example: One Time Pad. Not every application is compatible with our CKS notion and, hence, exhibits a CKS-compatible security game. In the following, we show that the one-time security game of the one-time pad (OTP) is incompatible. In other words, we show that winning the CKS-enhanced game is easier than winning the original game by a non-negligible gap.

Lemma 3. *Consider the one-time IND-CPA security game enhanced with a* Test *oracle When instantiated with the One Time Pad, there exists a PPT adversary $\mathcal{A}$ who wins this game with non-negligible advantage.*

Proof. Consider the following adversary: $\mathcal{A}$ chooses two distinct messages of appropriate length, m_0 and m_1, and submits them as the challenge. Given the challenge ciphertext c, the adversary queries the testing oracle for the key $k_0 \coloneqq c \oplus m_0$. If the testing oracle returns true, $\mathcal{A}$ guesses $b = 0$, and $b = 1$ otherwise. Since $c \coloneqq m_b \oplus k$, where k denotes the key used by the challenger, we can observe that $k_0 = c \oplus m_0 = k \oplus (m_b \oplus m_0)$ equals k iff $b = 0$. □

CKS-enhanced games. For a CKS-compatible game, we now define the enhanced game in which the adversary further gets to interact with parties executing the CKS scheme. Intuitively, we would hope that for any CKS-compatible game, this enhanced game is still secure. Unfortunately, this is impossible: Consider the game whose winning condition is to compute a valid CKS ciphertext, with the adversary in the original game having no information on the keys at all. Obviously, this game

Game $\mathsf{G}_{\mathsf{CKS}}$

```
Main
  b ← ⊥ ; n ← 0
  St[·], Msgs[·], RO[·] ← ⊥
  Known[·,·], ActualKey[·,·], Corr[·] ← false
  A^{Challenger, CreateUser, ..., RO}(1^κ)
  if b = ⊥ then b ← 0
  return b

Oracle Challenger
  req b = ⊥
  b ← 0
  (⊥; b) ← ⟨A ↔ C⟩

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

Oracle Corrupt
Input: u ∈ [n]
  for all e : ActualKey[u, e] = true do
    Corr[e] ← true
    C.Expose(e)
  return St[u]

Oracle Append
Input: (u, e, s, upload) ∈ [n] × ℕ × ({0,1}^κ ∪ {⊥}) × {0,1}
  req Known[u, e] = false
  if s = ⊥ then
    s ← C.Keys(e)
    // Record that party knows actual key
    ActualKey[u, e] ← true
  try (St[u], st_up) ← U.Append^RO(St[u], e, s, upload)
  if upload then
    try ⟨U.Upload^RO(st_up) ↔ A⟩
  Known[u, e] ← true

Oracle Erase
Input: (u, share) ∈ [n] × ℕ × ({0,1}^κ ∪ {⊥})
  try (St[u]; ⊥) ← ⟨U.Erase^RO(St[u], share) ↔ A⟩
  for e ∈ share do
    Known[u, e] ← false ; ActualKey[u, e] ← false

Oracle Grant
Input: (u, share, leak) ∈ [n] × P(ℕ) × {0,1}
  try (msg; ⊥) ← ⟨U.Grant^RO(St[u], share) ↔ A⟩
  if leak then
    for e ∈ share : ActualKey[u, e] do
      C.Expose(e)
    h ← msg
  else
    h ←$ {0,1}^κ // handle for delivery
    actual ← {e ∈ share | ActualKey[u, e]}
    Msgs[h] ← (msg, actual)
  return h

Oracle Accept
Input: (u', share, h, upload) ∈ [n] × P(ℕ) × {0,1}^* × {0.1}
  actual = ⊥
  if Msgs[h] ≠ ⊥ then
    (msg, actual) ← Msgs[h] // Delivery
  else
    msg ← h // Injection
  try (St[u'], st_up; ⊥) ← ⟨U.Accept^RO(St[u'], share, msg, upload) ↔ A⟩
  if upload then
    try ⟨U.Upload^RO(st_up) ↔ A⟩
  if actual ≠ ⊥ then
    for e ∈ actual do
      if Known[u', e] = false then
        ActualKey[u', e] ← true
  for e ∈ share do
    Known[u', e] ← true

Oracle Retrieve
Input: (u, share) ∈ [n] × P(ℕ)
  try (secrets; ⊥) ← ⟨U.Retrieve(St[u], share) ↔ A⟩

Random Oracle
Input: x ∈ {0,1}^*
  if RO[x] = ⊥ then RO[x] ←$ {0,1}^κ
  return RO[x]
```

Fig. 5: The CKS-enhanced game $\mathsf{G}_{\mathsf{CKS}}$ for a CKS-compatible one $\mathsf{G} = (\mathcal{C}, \alpha)$ and a CKS scheme CKS. The advantage function α is the same as the one associated with the underlying game G. The adversary is assumed to not invoke two oracles for the same user u concurrently.

transforms from impossible to trivial to win the moment the CKS-enhanced game provides CKS ciphertexts to the adversary.

Clearly, this is a contrived game and does not correspond to an application we actually care for. It shows, however, that we must enforce some level of separation between the application and the CKS scheme. (For applications we care for, such as SM, this is not an issue as those schemes have not been designed with the adversarial goal of breaking CKS.) We resort to the random oracle model and allow the CKS scheme to employ a fresh random oracle on which the game/application cannot depend.[13] We believe this to be a good compromise, admitting a broad class of applications to which CKS can be applied.

Definition 6. *For a CKS-compatible game* $\mathsf{G} = (\mathcal{C}, \alpha)$ *and a* $(\mathcal{G}, \mathcal{R}, \mathcal{E})$*-CKS scheme* CKS*, we define the enhanced CKS game* $\mathsf{G}_{\mathsf{CKS}} = (\mathcal{C}_{CKS}, \alpha)$ *with the following challenger:*

[13] While, for instance, a secure messaging protocol to be combined with CKS cannot depend on that random oracle, the messages sent (which are inputs to the security game) still could.

– *The challenger $\mathcal{C}_{CKS}$ internally runs $\mathcal{C}$ and provides all the public interface of $\mathcal{C}$ to the adversary (by forwarding the queries and responses).*
– *Additionally the challenger accepts the following queries simulating the execution of a CKS scheme: creating CKS users and appending sets on behalf of those users, as well as delegating and restoring keys.*
– *The adversary can moreover corrupt a CKS user, upon which $\mathcal{C}_{CKS}$ runs $\mathcal{C}.\mathsf{Expose}(\mathsf{e})$ for each epoch e known to the user before returning the user's CKS state. (This models that, using the leaked CKS state, the adversary can reconstruct those keys).*
– *The adversary can interact with the random oracle on which the CKS scheme* CKS *depends.*

A detailed description of $\mathcal{C}_{CKS}$ is given in Fig. 5, depicting the overall interaction between $\mathcal{C}_{CKS}$ and an adversary $\mathcal{A}$. We call an adversary $\mathcal{A}$ admissible *against $\mathcal{C}_{CKS}$ if the adversary does not interleave oracle calls for any user* u.[14]

We refer to Appendix C for further discussion on $\mathsf{G}_{\mathsf{CKS}}$. Using this game, we can now define CKS security for a scheme CKS. Simply put, the following definition requires that adding an execution of the scheme does not make winning any CKS-compatible game G substantially easier.

Definition 7. *A CKS scheme is secure if for any CKS-compatible game* G *we have that for any admissible PPT adversary $\mathcal{A}$ there exists a PPT adversary $\mathcal{A}'$, and a negligible function* $\mathsf{negl}\colon \mathbb{N} \to [0,1]$, *such that*

$$\mathsf{Adv}_{\mathsf{G}_{\mathsf{CKS}}}(\mathcal{A}) \leq \mathsf{Adv}_{\mathsf{G}}(\mathcal{A}') + \mathsf{negl}(\kappa).$$

4.4 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. See Fig. 6 for a formal definition of the integrity game, with the respective assertion in the Retrieve oracle. Let us briefly expand on some of the additional assertions: First, in the Append oracle the game asserts that a user rejects repeated appending unless it is the same key. Similarly, in the grant process, the granting user must reject if they do not know some of the keys, and the receiving party must reject if one of the keys mismatches a key they already know.

Note that in the case of a user u accepting an injected grant message msg, the game might not know u's resulting keys. Instead, the game assigns placeholders $\mathfrak{K}_p$, for $p \in \mathbb{N}$, that are (globally) replaced once the keys become known. This allows to enforce consistency even if u further delegates those keys. Additionally, we remark that we require integrity to hold even if the users' states are known to the adversary. This formalizes immediate post-compromise security (PCS) — ensuring that the PCS guarantees of the CKS scheme are at least as strong as the PCS guarantees of any scheme for which one intends to deploy CKS.

Definition 8. *We say that a CKS scheme* CKS *satisfies integrity if*

$$\Pr\left[\text{CKS-}\mathsf{Int}_{\mathsf{CKS}}^{\mathcal{A}} \Rightarrow 1\right] \leq \mathsf{negl}(\kappa)$$

for the game from Fig. 6 and any PPT adversary $\mathcal{A}$.

5 Compact Key Storage Schemes

In this section, we present two CKS schemes. Both of the schemes allow a party to share the entire communication history efficiently, with the former having constant local storage but linear key-recovery time, while the latter has both logarithmic local storage and key-recovery time. Additionally, the latter scheme also allows sharing and erasing arbitrary intervals of one's storage.

[14] It can be assumed that a single user would not engage in a CKS protocol interaction as long as they are still in an ongoing one.

Game $\text{CKS-Int}^{\mathcal{A}}_{\mathsf{CKS}}$ (Integrity)

```
Main
  win ← false
  n ← 0
  p ← 0
  St[·], Secret[·], Granted[·] ← ⊥
  A^{CreateUser,Append,Erase,Grant,Retrieve}(1^κ)
  return win

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

Oracle Append
Input: (u, e, k, upload) ∈ [n] × ℕ × {0,1}^κ × {0,1}
  try (St[u], st_up) ← U.Append(St[u], e, k, upload)
  if upload then
    try ⟨U.Upload(st_up) ↔ A⟩
  if Secret[u, e] ≠ ⊥ ∧ Secret[u, e] ≠ k
      ∧ ∀p ∈ ℕ : Secret[u, e] ≠ 𝔎_p then
    win ← true
  if ∃p ∈ ℕ : Secret[u, e] = 𝔎_p then
    for all (u′, e′) s.t. Secret[u′, e′] = 𝔎_p do
      Secret[u′, e′] ← k
  else
    Secret[u, e] ← k
  return St[u]

Oracle Erase
Input: (u, share) ∈ [n] × ℕ × 𝒫(ℕ)
  try (St[u]; ⊥) ← ⟨U.Erase(St[u], share) ↔ A⟩
  for e ∈ share do Secret[u, e] ← ⊥
  return St[u]

Oracle Grant
Input: (u, share) ∈ [n] × 𝒫(ℕ)
  try (msg; ⊥) ← ⟨U.Grant(St[u], share) ↔ A⟩
  for e ∈ share do
    if Secret[u, e] = ⊥ then win ← true
  Granted[msg] ← {(e, Secret[u, e]) | e ∈ share}
  return (St[u], msg)

Oracle Accept
Input: (u′, share, msg, upload) ∈ [n]×𝒫(ℕ)×{0,1}*×{0,1}
  try (St[u′], st_up; ⊥) ← ⟨U.Accept(St[u′], share, msg, upload) ↔ A⟩
  if upload then
    try (St[u′]; ⊥) ← ⟨U.Upload(st_up) ↔ A⟩
  if Granted[msg] ≠ ⊥ then
    if share ≠ {e | (e, ·) ∈ Granted[msg]} then
      win ← true
    for (e, k) ∈ Granted[msg] do
      if Secret[u′, e] ∉ {⊥, k} then win ← true
      Secret[u′, e] ← k
  else
    for e ∈ share do
      if Secret[u′, e] = ⊥ then
        p ← p + 1; Secret[u′, e] ← 𝔎_p
  return St[n′]

Oracle Retrieve
Input: (u, share) ∈ [n] × 𝒫(ℕ)
  try (secrets; ⊥) ← ⟨U.Retrieve(St[u], share) ↔ A⟩
  for e ∈ share do
    if Secret[u, e]≠secrets(e) ∧ ∀p ∈ ℕ : Secret[u, e]≠𝔎_p then
      win ← true
    if ∃p ∈ ℕ : Secret[u, e] = 𝔎_p then
      for all (u′, e′) s.t. Secret[u′, e′] = 𝔎_p do
        Secret[u′, e′] ← secrets(e)
  return St[u]
```

Fig. 6: 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., $\mathfrak{K}_p \neq \bot$, $\mathfrak{K}_p \neq \mathrm{k}$ for any bitstring k, and $\mathfrak{K}_p \neq \mathfrak{K}_q$ for $p \neq q$.

5.1 The Line Scheme

The first scheme is denoted $\mathsf{CKS}_{\mathsf{HA}}$ for history access. It allows delegating prefixes and efficiently retrieving consecutive intervals of keys[15]. More formally, its delegation predicate $\mathcal{G}_{\mathrm{HA}}$ allows parties to delegate arbitrary prefixes of the keys they know, and $\mathcal{R}_{\mathrm{HA}}(\mathsf{Known}, \mathrm{share})$ returns true iff share is a consecutive sequence of epochs for which the party knows all keys. The scheme does not offer forward secrecy, i.e., it does not allow selectively erasing data.

Recall the high-level overview of the scheme from Section 1.3. In a nutshell, the scheme leverages convergent encryption (CE) as follows: to outsource a new secret s_{e} when having state $\mathrm{st}_{\mathsf{e}-1} := (\mathsf{e}-1, K_{\mathsf{e}-1}, T_{\mathsf{e}-1})$ (where $K_{\mathsf{e}-1}$ is a symmetric key used to encrypt the secret for the previous epoch, and $T_{\mathsf{e}-1}$ is a tag to authenticate the corresponding ciphertext), a party first deterministically derives $K_{\mathsf{e}} := \mathrm{H}(\mathrm{s}_{\mathsf{e}} \| K_{\mathsf{e}-1})$ to then encrypt the pair under said key as $C_{\mathsf{e}} := \mathrm{SE.Enc}(K_{\mathsf{e}}, (\mathrm{s}_{\mathsf{e}} \| K_{\mathsf{e}-1}))$, for a suitable symmetric encryption scheme SE. To ensure integrity against a malicious server, the party moreover computes a tag $T_{\mathsf{e}} := \mathrm{H}(C_{\mathsf{e}} \| T_{\mathsf{e}-1})$ that is stored as part of the new state $\mathrm{st}_{\mathsf{e}} := (\mathsf{e}, K_{\mathsf{e}}, T_{\mathsf{e}})$ alongside the key. We dubbed this process "Derive" and the inverse operation "Invert," which are repeated for each new secret — see Fig. 2 for a graphical representation.

[15] An individual key for epoch e can, for instance, be retrieved as the singleton interval $[\mathsf{e}, \mathsf{e}]$ or as part of any interval containing e.

In the following, we discuss some omitted details such as appending secrets out-of-order, and how this ties in with the CKS operations such as access delegation. A formal description of the scheme is presented in Appendix E.1. A formal description of the scheme is presented in the full version.

Protocol state. As described above, the user stores the current CE key and CE tag. When appending an out-of-order secret, the user just stores the secret as part of its local state for the time being. More formally, let $\mathsf{e_{max}}$ be the largest epoch number such that the user learned all secrets $(s_1, \ldots, s_{\mathsf{e_{max}}})$. (We use $\mathsf{e_{max}}$ here to distinguish it from the epoch e and secret s_e input to U.Append.) The protocol state is then defined as $\mathrm{st} := (\mathsf{e_{max}}, K_{\mathsf{e_{max}}}, T_{\mathsf{e_{max}}}, \mathsf{Secret})$, where $\mathsf{Secret}[\mathsf{e}]$ stores the secret the user learned for any $\mathsf{e} > \mathsf{e_{max}} + 1$. Let ℓ denote the output length of the hash function. In U.Init, the state is initialized to $\mathsf{e_{max}} = 0$, $K_0 = 0^\ell$, $T_0 = 0^\ell$, and an empty mapping Secret. The server state $\mathrm{st_S}$, on the other hand, just consists of an (append-only) key-value storage. It maps T_e to pairs $(C_\mathsf{e}, T_{\mathsf{e}-1})$ consisting of the associated ciphertext and the tag of the prior epoch.

Outsourcing the keys. Assume the user has a state $\mathrm{st} := (\mathsf{e_{max}}, K_{\mathsf{e_{max}}}, T_{\mathsf{e_{max}}}, \mathsf{Secret})$ and learns the secret for the next consecutive epoch, i.e., $\mathsf{e} = \mathsf{e_{max}} + 1$. Then they invoke the "Derive" process to update their state and obtain c_e. If all secrets have been learned in order, then U.Append is done at this point. However, it could also be that the user already knows $s_{\mathsf{e}+1}$, i.e., $\mathsf{Secret}[\mathsf{e}+1] \neq \bot$. In this case, the algorithm simply runs "Derive" again and erases $\mathsf{Secret}[\mathsf{e}+1]$ afterward. This process is repeated until no further consecutive secret is known; we call this process state *compaction.* The upload state $\mathrm{st_{up}}$ output by U.Append is then the set of all ciphertexts $\{c_\mathsf{e}, \ldots, c_{\mathsf{e}'}\}$ produced in this process and U.Upload simply sends $\mathrm{st_{up}}$ to the server. Note that for each such ciphertext $c_\mathsf{e} := (\mathsf{e}, C_\mathsf{e}, T_{\mathsf{e}-1})$, S.Upload computes $T_\mathsf{e} := \mathrm{H}(C_\mathsf{e} \| T_{\mathsf{e}-1})$ and then stores the mapping from $(\mathsf{e}, T_\mathsf{e})$ to $(C_\mathsf{e}, T_\mathsf{e})$. The server computing the tag themselves is crucial for a malicious party not being able to overwrite honest user's ciphertexts.

Whenever U.Append receives a secret for an out-of-order epoch e with $\mathsf{e} > \mathsf{e_{max}} + 1$, it just stores s_e as part of their local state as $\mathsf{Secret}[\mathsf{e}] := s_\mathsf{e}$; nothing gets outsourced to the server at this point.

Retrieving keys. Now assume that a user with state $\mathrm{st} := (\mathsf{e_{max}}, K_{\mathsf{e_{max}}}, T_{\mathsf{e_{max}}}, \mathsf{Secret})$ wants to retrieve a key for epoch e. There are two cases: First, if $\mathsf{e} > \mathsf{e_{max}}$ then U.Retrieve just outputs $\mathsf{Secret}[\mathsf{e}]$, returning either the secret or an error in case this value has not been set. If $\mathsf{e} \leq \mathsf{e_{max}}$, then the user needs to recover the key using the help of the server. To this end, the user requests the ciphertexts $(c_\mathsf{e}, \ldots, c_{\mathsf{e_{max}}})$ from S.Retrieve and uses the "Invert" procedure to retrieve the keys. Note that, crucially, this process also works if another party U′ (or various parties) initially has outsourced the ciphertexts. Furthermore, observe that the user can request those ciphertexts simply by specifying the latest tag $T_{\mathsf{e_{max}}}$ and the number of ciphertexts they need. While the effort is linear in the number of epochs the user wants to go back, it is (asymptotically) optimal in recovering the keys for the entire suffix $[\mathsf{e}, \mathsf{e_{max}}]$.

Delegating access. If a user U_1 wishes to delegate access to the keys of epochs in share $= \{1, \ldots, \mathsf{e}\}$ to some other user U_2, they can send that user the state $(\mathsf{e}, K_\mathsf{e}, T_\mathsf{e})$. If $\mathsf{e} < \mathsf{e_{max}}$, then U_1 first recovers this state using "Invert" and the help of the server, analogous to when retrieving the secrets. Assume in the following that U_2 knows state $\mathrm{st}' := (\mathsf{e_{max}}', K_{\mathsf{e_{max}}'}, T_{\mathsf{e_{max}}'}, \mathsf{Secret}')$ for some $\mathsf{e_{max}}' < \mathsf{e}$ (as otherwise there would be no need for delegation). U_2 then first checks the consistency of $(\mathsf{e}, K_\mathsf{e}, T_\mathsf{e})$ with their local state. If $\mathsf{e_{max}}' > 0$, i.e., they have some non-trivial state, then they "Invert" from the received state until $\mathsf{e_{max}}'$ and check that they recover the same ciphertext and tag $K_\mathsf{e}, T_\mathsf{e}$). In addition, for any $\mathsf{e_{max}}' < \mathsf{e}' \leq \mathsf{e}$ such that $\mathsf{Secret}'[\mathsf{e}'] \neq \bot$ they check the consistency along the way and then erase it from Secret'. If $\mathsf{e_{max}}' = 0$, then they invert until the smallest epoch for which $\mathsf{Secret}'[\mathsf{e}'] \neq \bot$ and just check consistency with those keys. If all checks succeeded, they set $\mathrm{st}' := (\mathsf{e}, K_\mathsf{e}, T_\mathsf{e}, \mathsf{Secret}')$, with the pruned Secret'. Finally, the further compact their state in case $\mathsf{Secret}''[\mathsf{e}+1] \neq \bot$, potentially producing an upload state $\mathrm{st_{up}}$ for U_2.Upload.

Security and correctness. By inspection, we observe that correctness trivially follows. In particular, the collision resistance of the tagging mechanism (hash function) ensures that the honest server hands

the parties back the correct ciphertexts — even when malicious parties are present — while the determinism of the CE scheme further ensures the compactness of the server's state. Correctness of the recovered keys then follows from correctness of the CE scheme, which in turn follows from correctness of the underlying symmetric encryption scheme SE.

Theorem 1. *The line scheme* $\mathsf{CKS}_{\mathsf{HA}}$ *is correct, i.e., assuming the underlying* CE *scheme to be correct, we have for any PPT adversary* $\mathcal{A}$

$$\Pr\Big[(\mathcal{G}_{\mathrm{HA}}, \mathcal{R}_{\mathrm{HA}}, \mathcal{E}_{\mathrm{HA}})\text{-CKS-}\mathsf{Corr}^{\mathcal{A}}_{\mathsf{CKS}_{\mathsf{HA}}} \Rightarrow 1\Big] \leq \mathsf{negl}(\kappa).$$

For security, let us first consider integrity. Here, integrity mainly follows by collision resistance of H, and thus the locally stored T binding the entire history of ciphertexts. Combined with the correctness of the CE scheme, this ensures that parties recover the correct key (or there is a denial of service). A proof of the following theorems is presented in Appendix E.2.

Theorem 2. *The line scheme* $\mathsf{CKS}_{\mathsf{HA}}$ *satisfies integrity, i.e., for any PPT adversary* $\mathcal{A}$

$$\Pr\Big[\text{CKS-}\mathsf{Int}^{\mathcal{A}}_{\mathsf{CKS}_{\mathsf{HA}}} \Rightarrow 1\Big] \leq \mathsf{negl}(\kappa),$$

if the hash function H *is collision resistant and the* SE *scheme correct.*

Finally, consider CKS security. For our analysis, we require the convergent encryption to be *non-committing*. We discuss this in Appendix D in more detail. We remark that this stems from the strong CKS security notion that requires that if an application can tolerate fully adaptive corruption then adding CKS must preserve this property.[16] Notice however, that one-time security is sufficient for our cause: honest parties encrypt each distinct message with an independent key, as the encryption keys are deterministically derived as a hash of the ciphertext.

Theorem 3. *The line scheme* $\mathsf{CKS}_{\mathsf{HA}}$ *is secure, i.e., for any CKS-compatible game* G *and any admissible PPT adversary* $\mathcal{A}$ *there exists a PPT adversary* $\mathcal{A}'$ *such that*

$$\mathsf{Adv}_{\mathsf{G}_{\mathsf{CKS}}}(\mathcal{A}) \leq \mathsf{Adv}_{\mathsf{G}}(\mathcal{A}') + \mathsf{negl}(\kappa),$$

if the SE *scheme is correct, non-committing, and one-time IND-CPA secure, and the hash function is modeled as a random oracle.*

Efficiency. Our scheme supports efficient delegation of the entire prefix, as it sends a single ciphertext C_{e} and tag T_{e}, whose sizes do not depend on the current epoch. For retrieving secrets $[i, j]$, the scheme needs $\mathcal{O}(\mathsf{e}_{\mathsf{max}} - i)$ basic cryptographic operations if the party is currently in epoch $\mathsf{e}_{\mathsf{max}}$. This is optimal when retrieving the entire suffix and suboptimal when retrieving individual keys. Additionally, the same number of ciphertexts is needed from the server. The state is of constant size (in the number of epochs) if the party knows the keys of a consecutive prefix of the epochs. Otherwise, it additionally contains a number S of (disjoint) seeds. In the best case, appending involves a single CE encryption and outsourcing a single CE ciphertext, while worst case, it might involve up to $\mathrm{S} + 1$ such operations and ciphertexts — it is easy to see, however, that amortized it is $\mathcal{O}(1)$ encryptions.

Table 1 shows the concrete efficiency of appending the N-th secret to a CKS with $N-1$ consecutive secrets stored, with the primitives instantiated as follows. We instantiate H with a hash function with arbitrary input size and output length ℓ, i.e., $\mathrm{H}\colon \{0,1\}^* \to \ell$. We envision instantiating the deterministic one-time non-committing IND-CPA encryption using a hash-based Counter Mode: for each block of length ℓ, derive a pad by hashing the secret key and a counter, and XOR-ing the resulting steam with the plaintext.

[16] For applications that are selectively secure only, our CKS schemes when instantiating SE with a regular (deterministic) encryption scheme should suffice.

	# hashes	client state size	server upload size	server state size
line CKS	4	$2\ell + k$	$3\ell + k$	$N(4\ell + k)$
interval CKS, $N = 2^h$	$4h$	$2\ell + k$	$h(3\ell + k)$	$(N - 1)(5\ell + k)$
interval CKS, N is odd	0	extra $\ell + k$	0	unchanged

Table 1: Concrete efficiency of U.Append in our CKS schemes. We consider the cost of adding the N-th key to a CKS with $N - 1$ consecutive keys stored. ℓ denotes the hash output length, which is assumed to be the same as the length of a secret, while k denotes the space necessary to store the max epoch number, as well as node identifiers in the binary tree for the interval scheme. The first column refers to the number of invocations of the hash function, each with an input of size up to 3ℓ. We ignore the (small) overhead of maintaining an efficient dictionary for the server-stored ciphertexts and only count the space necessary to store keys and values.

Drawbacks. While the above scheme has several benefits — such as its simplicity or having constant local state and upload bandwidth per appended secret — it has some shortcomings. It does not provide efficient retrieval of individual keys (taking time $(n - j)$ to recover key j from state/key n) and has a coarse-grained delegation mechanism being limited to entire prefixes. In particular, it does not allow a party that has not been part of the CKS from the beginning to contribute, unless that party is granted access to the entire history first. This limits it to applications where it can be assumed that all parties know all keys. (There exists a number of applications such as Slack or Keybase where new group members get access to the entire chat history.)

Finally, even though the delegation of the latest prefix is efficient (the party can just send its current state) under certain conditions the process of accepting said delegation is not. Assume a party previously has known secrets 1 to i and at a later stage another party wants to delegate them access from 1 to j with $i \ll j$. Such a situation could, for instance, occur if the party has gone offline for an extended period of time or a device has been presumed lost and then found again. In such a case there is a subtle attack in which the granting party sends a valid CKS state that is inconsistent with the accepting party's prior keys. One might hope that, even if the granting party is malicious, the accepting party should not lose access to past keys (assuming the server cooperates) while the accepting party obviously still wants to compact its state. To this end, upon receiving the newer state the accepting party has to perform $\mathcal{O}(j - i)$ computation to verify that their prior state is consistent with the shared one.

5.2 The Interval Scheme

We now present a scheme called $\mathsf{CKS}_{\mathsf{Interval}}$, addressing the aforementioned drawbacks of the $\mathsf{CKS}_{\mathsf{HA}}$ scheme. The $\mathsf{CKS}_{\mathsf{Interval}}$ scheme enables a user Alice — currently located at epoch n — to help another user Bob to get enough information to retrieve keys in some interval $[i, j]$, where $i \leq j \leq n$. Furthermore, the scheme allows to efficiently retrieve or erase keys of any interval $[i, j]$. In particular, it enables parties to recover secrets they once knew in worst-case logarithmic time and allows for fine-grained delegation and erasure, efficiently supporting arbitrary intervals of one's storage.

On a high level, the scheme replaces the iterative Merkle-Damgård-esque approach with a Merkle-Tree-esque construction. That is, the construction assigns epochs to leaves in a binary tree. Instead of aggregating the previous state with the new epoch's secrets, the construction then either aggregates two secrets (at the leaves) or two states (for other nodes). See Fig. 7 for a graphical overview of the outsourcing process. More concretely, each node v in the tree has a state $\mathrm{st}_v = (K_v, T_v)$ consisting of a CE key and tag assigned. (Of course, the user's protocol state st will only store the node states for a minimal set of nodes.) Whenever the protocol knows the state for both children v_{left} and v_{right} of a node v, it then derives the state of v mostly analogous to the $\mathsf{CKS}_{\mathsf{HA}}$ scheme:

Derive$\big(\mathrm{st}_{v_{\mathsf{left}}} = (K_{v_{\mathsf{left}}}, T_{v_{\mathsf{left}}}),\ \mathrm{st}_{v_{\mathsf{right}}} = (K_{v_{\mathsf{right}}}, T_{v_{\mathsf{right}}})\big)$:

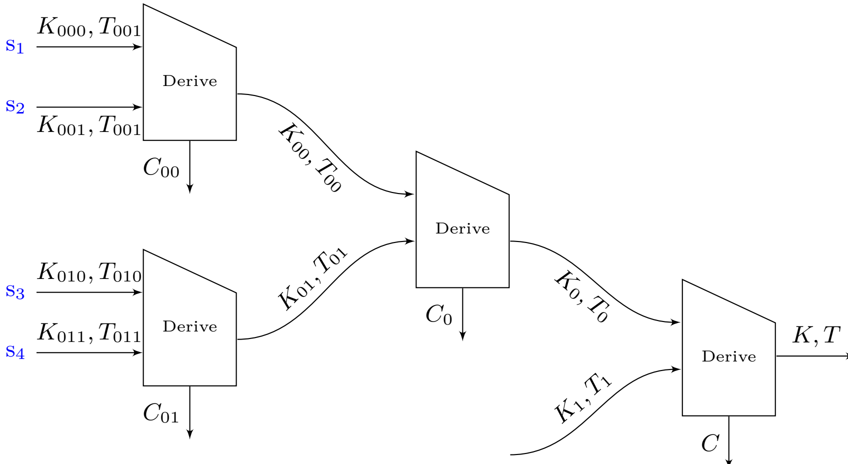

Fig. 7: A schematic representation of the Interval Scheme, showing the top half of the tree that supports four epochs. The solid lines show the "compaction" of the secrets (labeled by their epoch number) using convergent encryption, resulting in a CE ciphertext sent to the server and ciphertext and tag for the next level (labeled by the node index). At the leaves, the scheme sets the key to be equal to the secret (e.g., $K_{000} \coloneqq \mathrm{s}_1$) and the tag to be 0^ℓ.

1. $K_v \leftarrow \mathrm{H}(K_{v_{\mathsf{left}}} \| K_{v_{\mathsf{right}}})$
2. $C_v \leftarrow \mathrm{SE.Enc}(K_v, (K_{v_{\mathsf{left}}} \| K_{v_{\mathsf{right}}}))$
3. $T_{\mathsf{e}} \leftarrow \mathrm{H}(C_{\mathsf{e}} \| T_{v_{\mathsf{left}}} \| T_{v_{\mathsf{right}}})$
4. Sends $\mathrm{c}_v \leftarrow (v, C_v, T_{v_{\mathsf{left}}}, T_{v_{\mathsf{right}}})$ to the server and stores $\mathrm{st}_v \leftarrow (K_v, T_v)$.

For the e-th leaf, corresponding to epoch e, $\mathrm{st}_{v_{\mathsf{e}}} = (\mathrm{s}_{\mathsf{e}}, 0^\ell)$ is set to the secret s_{e} and a constant tag. The corresponding "Invert" procedure then recovers the states of the children of v:

Invert$\big(\mathrm{st}_v = (K_v, T_v),\ \mathrm{c}_v = (v, C_v, T_{v_{\mathsf{left}}}, T_{v_{\mathsf{right}}})\big)$:
1. $T'_v \leftarrow \mathrm{H}(C_v \| T_{v_{\mathsf{left}}} \| T_{v_{\mathsf{right}}})$;
2. Check $T'_v \stackrel{?}{=} T_v$;
3. $(K_{v_{\mathsf{left}}} \| K_{v_{\mathsf{right}}}) \leftarrow \mathrm{SE.Dec}(K_v, C_v)$;
4. Output $\mathrm{st}_{v_{\mathsf{left}}} \leftarrow (K_{v_{\mathsf{left}}}, T_{v_{\mathsf{left}}})$ and $\mathrm{st}_{v_{\mathsf{right}}} \leftarrow (K_{v_{\mathsf{right}}}, T_{v_{\mathsf{right}}})$.

Note that while non-trivial *compaction* in the line scheme was only necessary in case of out-of-order appending of secrets, in the tree whenever a secret is learned for some node v (e.g., for a leaf in case of U.Append or for an intermediate node as part of U.Accept) then compaction can potentially happen on the path from v to the root. Finally, observe that even for in-order appending the state of a party can become logarithmically sized, instead of constant as in the line scheme. In the following, we describe this state and the overall protocol in a bit more detail — see Appendix F.1 for a formal description of the scheme.

Protocol state. In the following, we describe the protocol that assumes some upper bound $T = 2^h$ on the number of epochs. Parties therefore store a binary tree τ of height h, with each node v having associated state $\mathrm{st}_v \coloneqq (K_v, T_v)$ consisting of a CE key and tag. For simplicity, we will write $v.K$ and $v.T$ to denote the respective components of st_v. Moreover, for simplicity, we assume that $\ell = \kappa$, and therefore the output length of the hash function is the same as the length of a secret. (Otherwise, an appropriate padding needs to be performed to secrets.) In slight abuse of notation, we, therefore, store $v_{\mathsf{e}}.K \coloneqq \mathrm{s}_{\mathsf{e}}$ and $v_{\mathsf{e}}.T \coloneqq 0^\ell$ at a leaf. The invariant the client state maintains is that on each path from a leaf to the root at most one node has values assigned. Consider a continuous interval of epochs $\mathcal{I}_j = [a, b] \subseteq [T]$. Observe that the minimal set of nodes such that exactly the leaves in $\mathcal{I}_j$ can be

reached from those nodes is of size $2\log(T)$. We call this set the *cover* of $\mathcal{I}_j$, denoted $\texttt{*cover}(\mathcal{I}_j)$. If $\mathcal{I}_1 \cup \cdots \cup \mathcal{I}_\eta$ denotes the partition of disjoint epoch intervals for which a party does "know" the secret, local storage st thus is of size $\mathcal{O}(\eta \cdot \log(T))$.[17]

The server, on the other hand, for each node v maintains a mapping from tag T_v to a ciphertext C_v and tags $T_{v_{\mathsf{left}}}$ and $T_{v_{\mathsf{right}}}$ of the left and right child nodes. Crucially, the server computes $T_v :=$ $\mathrm{H}(C_v \| T_{v_{\mathsf{left}}} \| T_{v_{\mathsf{right}}})$ themselves, to prevent malicious insider attacks.

Outsourcing the secrets. When U.Append gets a secret s_{e} for epoch e, it takes the e-th leaf v_{e} and assigns $v_{\mathsf{e}}.K = \mathrm{s}_{\mathsf{e}}$ and $v_{\mathsf{e}}.T = 0^\ell$. It then traverses the path from v_{e} to the root. For as long as the sibling node v_{sib} also has $v_{\mathsf{sib}}.K$ assigned, it *compacts* the state by using "Derive" to compute the state for the parent node while deleting the one for the children. The upload state $\mathrm{st}_{\mathrm{up}}$ then contains c_v of all nodes v for which "Derive" has been used to compute the state. U.Upload just sends $\mathrm{st}_{\mathrm{up}}$ to the server, which stores the ciphertexts by maintaining the mapping. For each append operation, the client thus has running time $\mathcal{O}(\log(T))$ and sends, if required, the same amount of information to the server.

Retrieving secrets. Recall that the protocol maintains the following invariant: for every epoch e for which s_{e} has been appended, there exists (exactly) one node v^* on the path from the corresponding leaf v_{e} to the root that has a CE key and tag stored. Hence, retrieving s_{e} requires walking up the tree until v^* is found, and then using "Invert" to recover the CE keys towards to root, for which $\mathrm{s}_{\mathsf{e}} = v_{\mathsf{e}}.K$. More generally, to recover the secrets of a continuous interval $[i, j]$, U.Retrieve computes $\texttt{*cover}([i, j])$ and for each $v \in \texttt{*cover}([i, j])$ finds the node v^* along the path from v to the root that has its state assigned. From those nodes, all secrets in $[i, j]$ can then be recovered using "Invert" as described above. This leads to an overall complexity of $\mathcal{O}((j - i) \cdot \log(T/(j - i)))$, which also corresponds to the communication complexity with the server.

Delegating access. To delegate access to a continuous interval $[i, j]$ of secrets, U_1.Grant sends the states for all nodes in $\texttt{*cover}([i, j])$ as msg. Similar to the last paragraph, the algorithm might first have to re-derive those keys using "Invert" and the ciphertexts from the server. U_2.Accept then (if required) checks the consistency of msg with any secrets in $[i, j]$ they already know. If all checks succeed, they erase the parts they verified and instead store the states for $\texttt{*cover}([i, j])$ they received. Finally, they compact their state, potentially outputting $\mathrm{st}_{\mathrm{up}}$.

Note that msg consists of $\mathcal{O}(\log(j - i)) \subseteq \mathcal{O}(\log(T))$ elements. Furthermore, a moment of reflection shows that the consistency checks by U_2.Accept takes at most $\mathcal{O}(\eta \cdot \log(T))$ time, where η refers to the number of currently stored disjoint intervals.

Erasing keys. To erase a secret s_{e} for epoch e, a party needs to erase all states along the path from v_{e} to the root. To this end, U.Erase finds the node v along this path for which it stores the state. Then it expands v using "Invert" and the ciphertext c_v from the server, erasing the state of v and storing the one for v_{left} and v_{right} instead. Then they recurse on the child (e.g., v_{left}) that is on the path to v_{e}. More generally, erasing an interval $[i, j]$ can be efficiently implemented in time $\mathcal{O}((j - i) \cdot \log(T/(j - i)))$.

Security and correctness. Correctness follows analogously to the line scheme by inspection. In particular, the correctness of the CE scheme implies $\texttt{*recover}$ correctly recovering CE keys and tags for inner nodes and, finally, secrets on the leaf level. Note that the server indexing ciphertexts by tag moreover guarantees that each party gets the correct one even if malicious insiders interact with the server, assuming the hash function used for the tag computation is collision-resistant. This is summarized in the following theorem.

Theorem 4. *The interval scheme* $\mathsf{CKS}_{\mathsf{Interval}}$ *is correct if the underlying* CE *scheme is correct and the hash function used in* CE. Tag *is collision resistant.*

[17] We assume the tree to be compactly represented such that storage grows only in the number of vertices with assigned properties.

Security also follows along a similar vein as for the line scheme: tags ensure integrity while (one-time) IND-CPA security of the CE scheme ensures confidentiality.

Theorem 5. *Assuming the* CE *scheme is correct and one-time IND-CPA secure and modeling* H *as a random oracle, the interval scheme* $\mathsf{CKS}_{\mathsf{Interval}}$ *satisfies integrity and is secure. That is, for any CKS-compatible game* G*, and any admissible PPT adversary* $\mathcal{A}$*, there exists a PPT adversary* $\mathcal{A}'$ *such that*

$$\mathsf{Adv}_{\mathsf{G}_{\mathsf{CKS}}}(\mathcal{A}) \leq \mathsf{Adv}_{\mathsf{G}}(\mathcal{A}') + \mathsf{negl}(\kappa).$$

A proof of the theorem is presented in Appendix F.2.

Efficiency. As argued above, all operations run in (poly-)logarithmic time, except for retrieving n consecutive keys, which has run time $\mathcal{O}(n \cdot \log(T/n))$, and accepting a delegation of n consecutive secrets, which runs in time $\mathcal{O}(n)$. The latter further requires a message of size $\mathcal{O}(\log n)$ to be sent over a secure out-of-band channel. Communication complexity with the server typically corresponds to the running time and at each point in time a party stores at most $\mathcal{O}(\log T)$ local state.[18]

When instantiating the primitives in the same way as for the line scheme, Table 1 shows the concrete efficiency of appending the N-th secret to a CKS with $N-1$ consecutive secrets stored in two cases. First, we consider the scenario of $N = 2^h$, i.e., appending the very last secret while knowing secrets $1, \ldots, N-1$. In that case, U.Append can compact the entire tree into a single state for the root. Second, we consider the case of appending the secret of an odd epoch, in which case U.Append cannot compact anything.

References

1. Martín Abadi, Dan Boneh, Ilya Mironov, Ananth Raghunathan, and Gil Segev. Message-locked encryption for lock-dependent messages. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 374–391. Springer, Heidelberg, August 2013.
2. Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158. Springer, Heidelberg, May 2019.
3. Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 248–277. Springer, Heidelberg, August 2020.
4. Frederik Armknecht, Ludovic Barman, Jens-Matthias Bohli, and Ghassan O. Karame. Mirror: Enabling proofs of data replication and retrievability in the cloud. In Thorsten Holz and Stefan Savage, editors, *USENIX Security 2016*, pages 1051–1068. USENIX Association, August 2016.
5. Gilad Asharov, Ilan Komargodski, Wei-Kai Lin, Kartik Nayak, Enoch Peserico, and Elaine Shi. Optorama: Optimal oblivious RAM. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 403–432. Springer, 2020.
6. Giuseppe Ateniese, Randal Burns, Reza Curtmola, Joseph Herring, Osama Khan, Lea Kissner, Zachary Peterson, and Dawn Song. Remote data checking using provable data possession. *ACM Transactions on Information and System Security (TISSEC)*, 14(1):1–34, 2011.
7. Ali Bagherzandi, Stanislaw Jarecki, Nitesh Saxena, and Yanbin Lu. Password-protected secret sharing. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011*, pages 433–444. ACM Press, October 2011.
8. R. Barnes, B. Beurdouche, , J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. The messaging layer security (mls) protocol (draft-ietf-mls-protocol-latest). Technical report, IETF, Oct 2020. `https://messaginglayersecurity.rocks/mls-protocol/draft-ietf-mls-protocol.html`.
9. Mihir Bellare and Sriram Keelveedhi. Interactive message-locked encryption and secure deduplication. In Jonathan Katz, editor, *PKC 2015*, volume 9020 of *LNCS*, pages 516–538. Springer, Heidelberg, March / April 2015.

[18] The analysis depends on the binary tree being implemented in a space-efficient manner while still allowing for constant-time tree traversal.

10. Mihir Bellare, Sriram Keelveedhi, and Thomas Ristenpart. Message-locked encryption and secure deduplication. In Thomas Johansson and Phong Q. Nguyen, editors, *EUROCRYPT 2013*, volume 7881 of *LNCS*, pages 296–312. Springer, Heidelberg, May 2013.
11. Mihir Bellare, Igors Stepanovs, and Stefano Tessaro. Contention in cryptoland: Obfuscation, leakage and UCE. In Eyal Kushilevitz and Tal Malkin, editors, *TCC 2016-A, Part II*, volume 9563 of *LNCS*, pages 542–564. Springer, Heidelberg, January 2016.
12. John Best, Wayne Hineman, Steven Hetzler, Guerney Hunt, and Charanjit S. Jutla. Secure storage with deduplication. Cryptology ePrint Archive, Paper 2022/553, 2022. `https://eprint.iacr.org/2022/553`.
13. Alexander Bienstock, Yevgeniy Dodis, and Kevin Yeo. Forward secret encrypted RAM: Lower bounds and applications. Cryptology ePrint Archive, Report 2021/244, 2021. `https://eprint.iacr.org/2021/244`.
14. Manuel Blum, William S. Evans, Peter Gemmell, Sampath Kannan, and Moni Naor. Checking the correctness of memories. In *32nd FOCS*, pages 90–99. IEEE Computer Society Press, October 1991.
15. Dan Boneh, Saba Eskandarian, Sam Kim, and Maurice Shih. Improving speed and security in updatable encryption schemes. In *International Conference on the Theory and Application of Cryptology and Information Security*, pages 559–589. Springer, 2020.
16. Dan Boneh, Kevin Lewi, Hart William Montgomery, and Ananth Raghunathan. Key homomorphic PRFs and their applications. In Ran Canetti and Juan A. Garay, editors, *CRYPTO 2013, Part I*, volume 8042 of *LNCS*, pages 410–428. Springer, Heidelberg, August 2013.
17. Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 280–300. Springer, Heidelberg, December 2013.
18. Colin Boyd, Gareth T. Davies, Kristian Gjøsteen, and Yao Jiang. Fast and secure updatable encryption. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 464–493. Springer, Heidelberg, August 2020.
19. Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 501–519. Springer, Heidelberg, March 2014.
20. Christina Brzuska, Pooya Farshim, and Arno Mittelbach. Indistinguishability obfuscation and UCEs: The case of computationally unpredictable sources. In Juan A. Garay and Rosario Gennaro, editors, *CRYPTO 2014, Part I*, volume 8616 of *LNCS*, pages 188–205. Springer, Heidelberg, August 2014.
21. Dwaine E. Clarke, G. Edward Suh, Blaise Gassend, Ajay Sudan, Marten van Dijk, and Srinivas Devadas. Towards constant bandwidth overhead integrity checking of untrusted data. In *2005 IEEE Symposium on Security and Privacy*, pages 139–153. IEEE Computer Society Press, May 2005.
22. Reza Curtmola, Osama Khan, Randal Burns, and Giuseppe Ateniese. MR-PDP: Multiple-replica provable data possession. In *2008 the 28th international conference on distributed computing systems*, pages 411–420. IEEE, 2008.
23. Poulami Das, Julia Hesse, and Anja Lehmann. DPaSE: Distributed password-authenticated symmetric-key encryption, or how to get many keys from one password. In Yuji Suga, Kouichi Sakurai, Xuhua Ding, and Kazue Sako, editors, *ASIACCS 22*, pages 682–696. ACM Press, May / June 2022.
24. Gareth T. Davies, Sebastian Faller, Kai Gellert, Tobias Handirk, Julia Hesse, Máté Horváth, and Tibor Jager. Security analysis of the WhatsApp end-to-end encrypted backup protocol. In Helena Handschuh and Anna Lysyanskaya, editors, *Advances in Cryptology – CRYPTO 2023*, pages 330–361, Cham, 2023. Springer Nature Switzerland.
25. Giovanni Di Crescenzo, Niels Ferguson, Russell Impagliazzo, and Markus Jakobsson. How to forget a secret. In *Proceedings of the 16th Annual Conference on Theoretical Aspects of Computer Science*, STACS'99, page 500–509, Berlin, Heidelberg, 1999. Springer-Verlag.
26. Yevgeniy Dodis, Daniel Jost, Balachandar Kesavan, and Antonio Marcedone. End-to-end encrypted zoom meetings: Proving security and strengthening liveness. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 157–189. Springer, Heidelberg, April 2023.
27. J.R. Douceur, A. Adya, W.J. Bolosky, P. Simon, and M. Theimer. Reclaiming space from duplicate files in a serverless distributed file system. In *Proceedings 22nd International Conference on Distributed Computing Systems*, pages 617–624, 2002.
28. C Chris Erway, Alptekin Küpçü, Charalampos Papamanthou, and Roberto Tamassia. Dynamic provable data possession. *ACM Transactions on Information and System Security (TISSEC)*, 17(4):1–29, 2015.
29. Adam Everspaugh, Kenneth G. Paterson, Thomas Ristenpart, and Samuel Scott. Key rotation for authenticated encryption. In Jonathan Katz and Hovav Shacham, editors, *CRYPTO 2017, Part III*, volume 10403 of *LNCS*, pages 98–129. Springer, Heidelberg, August 2017.

30. Ben Fisch. Tight proofs of space and replication. In *Annual International Conference on the Theory and Applications of Cryptographic Techniques*, pages 324–348. Springer, 2019.
31. Ben Fisch, Joseph Bonneau, Nicola Greco, and Juan Benet. Scaling proof-of-replication for filecoin mining. *Benet//Technical report, Stanford University*, 2018.
32. Oded Goldreich and Rafail Ostrovsky. Software protection and simulation on oblivious RAMs. *Journal of the ACM (JACM)*, 43(3):431–473, 1996.
33. Michael T Goodrich and Michael Mitzenmacher. Privacy-preserving access of outsourced data via oblivious RAM simulation. In *International Colloquium on Automata, Languages, and Programming*, pages 576–587. Springer, 2011.
34. Shai Halevi, Danny Harnik, Benny Pinkas, and Alexandra Shulman-Peleg. Proofs of ownership in remote storage systems. In Yan Chen, George Danezis, and Vitaly Shmatikov, editors, *ACM CCS 2011*, pages 491–500. ACM Press, October 2011.
35. S. Jarecki, A. Kiayias, H. Krawczyk, and J. Xu. 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)*, pages 276–291, Los Alamitos, CA, USA, mar 2016. IEEE Computer Society.
36. Stanislaw Jarecki, Hugo Krawczyk, and Jason K. Resch. Updatable oblivious key management for storage systems. In Lorenzo Cavallaro, Johannes Kinder, XiaoFeng Wang, and Jonathan Katz, editors, *ACM CCS 2019*, pages 379–393. ACM Press, November 2019.
37. Ari Juels and Burton S. Kaliski Jr. Pors: proofs of retrievability for large files. In Peng Ning, Sabrina De Capitani di Vimercati, and Paul F. Syverson, editors, *ACM CCS 2007*, pages 584–597. ACM Press, October 2007.
38. Aggelos Kiayias, Stavros Papadopoulos, Nikos Triandopoulos, and Thomas Zacharias. Delegatable pseudorandom functions and applications. In Ahmad-Reza Sadeghi, Virgil D. Gligor, and Moti Yung, editors, *ACM CCS 2013*, pages 669–684. ACM Press, November 2013.
39. Michael Klooß, Anja Lehmann, and Andy Rupp. (R)CCA secure updatable encryption with integrity protection. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 68–99. Springer, Heidelberg, May 2019.
40. Anja Lehmann and Björn Tackmann. Updatable encryption with post-compromise security. In Jesper Buus Nielsen and Vincent Rijmen, editors, *EUROCRYPT 2018, Part III*, volume 10822 of *LNCS*, pages 685–716. Springer, Heidelberg, April / May 2018.
41. Daniel E. Lucani, Lars Nielsen, Claudio Orlandi, Elena Pagnin, and Rasmus Vestergaard. Secure generalized deduplication via multi-key revealing encryption. In Clemente Galdi and Vladimir Kolesnikov, editors, *SCN 20*, volume 12238 of *LNCS*, pages 298–318. Springer, Heidelberg, September 2020.
42. M. Marlinspike and T. Perrin. The double ratchet algorithm, 11 2016. `https://whispersystems.org/docs/specifications/doubleratchet/doubleratchet.pdf`.
43. Alina Oprea and Michael K. Reiter. Integrity checking in cryptographic file systems with constant trusted storage. In Niels Provos, editor, *USENIX Security 2007*. USENIX Association, August 2007.
44. Sarvar Patel, Giuseppe Persiano, Mariana Raykova, and Kevin Yeo. Panorama: Oblivious RAM with logarithmic overhead. In *2018 IEEE 59th Annual Symposium on Foundations of Computer Science (FOCS)*, pages 871–882. IEEE, 2018.
45. Benny Pinkas and Tzachy Reinman. Oblivious RAM revisited. In *Annual cryptology conference*, pages 502–519. Springer, 2010.
46. Hovav Shacham and Brent Waters. Compact proofs of retrievability. *Journal of Cryptology*, 26(3):442–483, July 2013.
47. Daniel Slamanig and Christoph Striecks. Revisiting updatable encryption: Controlled forward security, constructions and a puncturable perspective. Cryptology ePrint Archive, Paper 2021/268, 2021. `https://eprint.iacr.org/2021/268`.
48. Emil Stefanov, Marten Van Dijk, Elaine Shi, Christopher Fletcher, Ling Ren, Xiangyao Yu, and Srinivas Devadas. Path ORAM: an extremely simple oblivious RAM protocol. In *Proceedings of the 2013 ACM SIGSAC conference on Computer & communications security*, pages 299–310, 2013.
49. WhatsApp. How WhatsApp enables multi-device capability, 2021. Retrieved 10/2022 from `https://engineering.fb.com/2021/07/14/security/whatsapp-multi-device/`.

A Additional Preliminaries

Notation. We use $\mathbb{N} := \{1, 2, \ldots\}$ and for $x, y \in \mathbb{N}$ we write $[x] := \{1, 2, \ldots, x\}$ and $[x, y] := \{x, x+1, \ldots, y\}$. $x \leftarrow a$ 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.

Cryptographic Primitives. We use some very basic cryptographic primitives, such as a length-doubling pseudo-random generator $\mathrm{PRG}\colon \{0,1\}^\kappa \to \{0,1\}^{2\kappa}$, a (one-time) symmetric encryption scheme $\mathrm{SE} := (\mathrm{SE.Enc}, \mathrm{SE.Dec})$, and a hash function $\mathrm{H}\colon \{0,1\}^* \to \{0,1\}^\kappa$. For encryption we assume (one-time) IND-CPA security and for the hash function typically assume it to be a random oracle.

Binary Trees. One of our constructions makes use of a (complete) binary tree of height h, having 2^h leaves. We use standard object-oriented notation such as $v.\mathsf{parent}$ for tree traversal and manipulation. Let v_{root} denote the root node. Then, let the *direct path* of a leaf v denote the ordered sequence of nodes from v to v_{root}. The *copath* (or sibling path) of v denotes the sequence containing all the sibling nodes of the direct path. Finally, we denote by the left and right copaths the sequence of nodes when only considering those elements in the copath that are left children or right children, respectively.

B On CKS-compatible Games

In this section, we expand on the CKS compatibility of various security games. First, we formalize the intuition that when an application uses an independent instance of a cryptographic primitive per epoch, that then it suffices to show the CKS compatibility of a single instance. Second, we provide some further examples of the CKS compatibility of fundamental cryptographic primitives. Finally, we briefly sketch the CKS comparability of the Signal protocol.

B.1 Single to Many Instances

The examples we consider here are, for simplicity, CKS-compatible games where each epoch behaves independently. For instance, in the case of IND-CPA secure symmetric encryption, we consider the CKS-compatible game that for each epoch chooses an independent key and offers a respective challenge oracle. (More complex games one might be interested in, such as CGKA games, often behave almost independently, meaning that their epochs behave independently as long as for instance no corruption occurred.) As a result, we introduce the simplified notion of single-instance CKS compatibility.

Definition 9. *We say that a game $\mathsf{G}_1 = (\mathcal{C}_1, \alpha_1)$ is* single-instance *CKS-compatible if $\mathcal{C}$ provides a private oracle* Keys *oracle that outputs a (single) key $k \in \{0,1\}^\kappa$ or an error $\bot$, and an* Expose *that leaks the key k. (Those oracles take unary trigger inputs.)*

One can show that CKS compatibility is implied by single-instance CKS compatibility, as expressed in the following two results.

Lemma 4. *Let $\mathsf{G}_1 = (\mathcal{C}_1, \alpha_1)$ denote a single-instance CKS-compatible game for which $\alpha_1 := 0$. Now consider CKS-compatible game $\mathsf{G} = (\mathcal{C}, \alpha)$ obtained by the parallel composition — running one instance of G_1 per epoch — which is considered to be won if at least one of the instances is won (and for which $\alpha := 0$).*

For any PPT adversary $\mathcal{A}$ there exists an appropriately modified PPT adversary $\mathcal{A}'$ of roughly the same running time as $\mathcal{A}$ such that

$$\mathsf{Adv}_{\mathsf{G}}(\mathcal{A}) \leq N \cdot \mathsf{Adv}_{\mathsf{G}_1}(\mathcal{A}')$$

where N denotes an upper bound on the number of epochs $\mathcal{A}$ interacts with.

Proof. Consider the reduction $\mathcal{A}'$ that chooses an index $i \in [N]$ uniformly at random. It then internally runs $\mathcal{A}$ and uses its challenge instance for the i-th epoch $\mathcal{A}$ interacts with, and emulates the remaining instances itself.

Clearly, $\mathcal{A}'$ emulates G perfectly towards $\mathcal{A}'$, except for the winning condition. (Observe that G is won if any instance is won, but the emulated game is only won if the the challenge instance i is won.) Since however i is chosen uniformly at random, and does not affect the observable behavior of the emulated game, we have that the instance $\mathcal{A}$ wins is equal to i with at least probability $\frac{1}{N}$, inducing the claim.

Lemma 5. *Let $\mathsf{G}_1 = (\mathcal{C}_1, \alpha_1)$ denote a distinguishing single-instance CKS-compatible game, i.e., one for which the game internally chooses a bit b uniformly at random, is won whenever the adversary guesses b correctly, and has $\alpha_1 := \frac{1}{2}$.*

Now consider CKS-compatible game obtained by the parallel composition — running one instance of G_1 per epoch — with all instances sharing the same bit b, and $\alpha := \frac{1}{2}$. For any PPT adversary $\mathcal{A}$ there exists an appropriately modified PPT adversary $\mathcal{A}'$ of roughly the same running time as $\mathcal{A}$ such that

$$\mathsf{Adv}_{\mathsf{G}}(\mathcal{A}) \leq N \cdot \mathsf{Adv}_{\mathsf{G}_1}(\mathcal{A}')$$

where N denotes an upper bound on the number of epochs $\mathcal{A}$ interacts with.

Proof. Consider the reduction $\mathcal{A}^i$ that interacts with the challenger $\mathcal{C}_1$ as follows: It internally runs $\mathcal{A}$. For epochs 1 to $i-1$, it emulates an instance of G_1 with $b = 0$, uses the challenger its interacting with as the i-th instance, and emulates the remaining epochs $e > i$ with $b = 1$. We observe that for $i = 1$ and G_1 using $b = 0$, the emulated game exactly corresponds to G with $b = 0$. Analogously, for $i = N$ and $b = 1$ it corresponds to G with $b = 1$. Further, let $\mathcal{A}'$ be the adversary that chooses $i \in [N]$ uniformly at random and then runs $\mathcal{A}^i$.

Now, let b' denote the adversary's guess for b. Using the well-known correspondence of the distinguishing advantage, and the triangle inequality, we obtain

$$\begin{aligned}
&\mathsf{Adv}_{\mathsf{G}}(\mathcal{A}) \\
&= \left|\Pr[b' = 1 \mid \mathcal{A}(1^\kappa) \leftrightarrow \mathcal{C}(1^\kappa) \mid b = 1] - \Pr[b' = 1 \mid \mathcal{A}(1^\kappa) \leftrightarrow \mathcal{C}(1^\kappa) \mid b = 0]\right| \\
&\leq \sum_{i=1}^{N} \left|\Pr[b' = 1 \mid \mathcal{A}^i(1^\kappa) \leftrightarrow \mathcal{C}_1(1^\kappa) \mid b = 1] - \Pr[b' = 1 \mid \mathcal{A}^i(1^\kappa) \leftrightarrow \mathcal{C}_1(1^\kappa) \mid b = 0]\right| \\
&\leq N \cdot \left|\Pr[b' = 1 \mid \mathcal{A}'(1^\kappa) \leftrightarrow \mathcal{C}_1(1^\kappa) \mid b = 1] - \Pr[b' = 1 \mid \mathcal{A}'(1^\kappa) \leftrightarrow \mathcal{C}_1(1^\kappa) \mid b = 0]\right| \\
&= N \cdot \mathsf{Adv}_{\mathsf{G}_1}(\mathcal{A}'),
\end{aligned}$$

concluding the claim.

B.2 CKS Compatibility of Cryptographic Primitives

Pseudorandom Functions. In the following we focus on the core issue of the testing oracle and consider primitives with a single secret, the structural example can be obtained by considering one independent instance of the primitive per epoch. Moreover, we only allow for a single call to the testing oracle, which is equivalent to many calls up to a polynomial factor. Let us consider a PRF scheme. Intuitively, any adversary who can extract the PRF key by making simple queries can trivially distinguish its outputs from uniform random values.

Lemma 6. *Let G_{PRF} denote the standard PRF security game that lets the adversary either interact with the PRF ($b = 0$) or a uniform random function ($b = 1$). Let G^{+}_{PRF} denote the same game that exposes a testing oracle for the key k. For every PPT adversary $\mathcal{A}$ that can win G^{+}_{PRF} with non-negligible probability, there exists a PPT adversary $\mathcal{B}$ that wins G_{PRF} with non-negligible probability.*

Proof. Consider a hybrid game H_{PRF} that augments G_{PRF} by a trivial testing oracle that always returns 0. As this oracle can be emulated, winning this game is just as hard as winning the standard PRF security game G_{PRF}. To see that any PPT adversary $\mathcal{A}$ has the same winning probability for the hybrid as the game G^{+}_{PRF} with a proper testing oracle, we use the following reduction to G_{PRF}:

- The adversary $\mathcal{B}$ runs $\mathcal{A}$ internally, forwarding all evaluation queries to the PRF as well as the respective results.
- Once $\mathcal{A}$ calls the testing oracle on k', $\mathcal{B}$ chooses a fresh $\tilde{x}$ that has not been queried yet. It then queries G_{PRF} on $\tilde{x}$.
- It compare the result $\tilde{y}$ with $PRF(k', \tilde{x})$. if they match, return $b' = 0$ (real), otherwise sample b' uniformly at random.

Assume $\mathcal{A}$ distinguishes G^{+}_{PRF} and H_{PRF} with probability β. Since the games are equivalent-until-bad, this means $\mathcal{A}$ queries the testing oracle with $k' = k$ with probability (at least) β. Let γ denote the probability of the check in the reduction succeeding. If $b = 1$, then γ is negligible and, therefore, $\mathcal{B}$ guesses $b' = 1$ with probability at least $1/2 - \epsilon$, where $\epsilon \in \mathsf{negl}(\kappa)$. If $b = 0$, then we know that $\gamma \geq \beta$ (the check could still succeed even if the key differs e.g. in one bit). The conditional probability of $\mathcal{B}$ correctly guessing $b' = 0$ is $\gamma/2 + 1/2 \geq \beta/2 + 1/2$, leading to a non-negligible overall advantage. ☐

IND-CPA Secure Encryption. While one-time IND-CPA secure schemes (such as the One Time Pad) are not necessarily CKS compatible, regular encryption is. Intuitively, if an adversary could extract a candidate for the secret key after observing n ciphertexts, then they could try to leverage this guess to break the privacy of one more ciphertext.

Lemma 7. *Every symmetric-key encryption scheme* $\mathrm{SE} \coloneqq (\mathrm{SE.Enc}, \mathrm{SE.Dec})$ *satisfying the standard IND-CPA security* $\mathsf{G}_{\textsf{IND-CPA}}$ *notion for an exponential message space* $\mathcal{M}$ *also satisfies a single-instance CKS-compatible one* $\mathsf{G}'_{\textsf{IND-CPA}}$ *for which the challenger works as follows:*

- *The* Encrypt *oracle allows for arbitrary many encryptions, while the* Challenge *oracle allows for a single challenge with two messages of equal length.*
- *The* Expose *and* Test *oracles work as described by the respective definitions of a CKS-compatible game.*
- *Upon submitting the guess* b' *for the challenge bit* b*, the game outputs* $b = b'$ *if the key has not been exposed, and a uniform random winning bit otherwise.*

For any PPT adversary $\mathcal{A}$ *that has non-negligible advantage* $\mathsf{Adv}_{\mathsf{G}'_{\textsf{IND-CPA}}}(\mathcal{A})$*, there exists an appropriately modified PPT adversary* $\mathcal{A}'$ *of roughly the same running time as* $\mathcal{A}$ *that has non-negligible advantage* $\mathsf{Adv}_{\mathsf{G}_{\textsf{IND-CPA}}}(\mathcal{A}')$*.*

Proof. Consider a hybrid game $\mathsf{G}''_{\textsf{IND-CPA}}$ that functions like $\mathsf{G}'_{\textsf{IND-CPA}}$ but for which the Test returns 0 as long as the corresponding key has not been exposed. Afterwards, i.e, once the true key has been revealed to the adversary, the Test does the proper check. Observe that $\mathsf{G}'_{\textsf{IND-CPA}}$ and $\mathsf{G}''_{\textsf{IND-CPA}}$ are game-equivalent, i.e., they behave identical until the adversary queries the Test with the correct key when it has not yet been exposed. Hence, by the fundamental lemma of game playing, we can bound the difference of the winning advantages with the probability of triggering this bad event (guessing a key) in either of the games. (The winning probability of $\mathsf{G}''_{\textsf{IND-CPA}}$ can then be easily bounded by an appropriate reduction to IND-CPA.)

We proceed by bounding the advantage of triggering said condition in $\mathsf{G}''_{\textsf{IND-CPA}}$. More concretely, we argue that guessing a key actually implies breaking IND-CPA security, using the following reduction: Encryption and exposure queries are forwarded to the underlying game. The Test is perfectly emulated by always returning 0 unless the respective epoch has been exposed — in which case the reduction can execute the corresponding equality check. To answer a challenge on a pair m_0 and m_1, the reduction internally chooses a bit b'' uniformly at random and submits an encryption query of $m_{b''}$ instead. Note

that the reduction perfectly emulates the challenger of $\mathsf{G}''_{\mathsf{IND\text{-}CPA}}$ as in particular using the challenger's bit b'' instead of the underlying game's bit b does not affect the observable behavior.

Once the adversary finishes the interaction by submitting a guess b', the reduction proceeds by submits a challenge m'_0 and m'_1, where the two messages are chosen uniformly at random from the message space. The reduction then takes all the Test queries and tries to decrypts the respective challenge ciphertexts c using the submitted key k'. If this results in m'_0, then the reduction guesses $b = 0$ for the underlying game; otherwise, it guesses $b = 1$. It remains to argue about the reduction's performance. Assume that the adversary triggers the bad event, i.e., submits a non-trivial guess to the Test oracle. If $b = 0$, then by correctness of the encryption scheme c correctly decrypts to m'_0 with overwhelming probability. If, on the other hand, $b = 1$ then we argue that $\mathrm{SE.Dec}(k', c) \neq m'_0$. To this end, observe that both k and c are statistically independent of m'_0, as the k' was submitted before m'_0 was even sampled and the encryption scheme received m'_1 instead as part of the challenge. Hence, if $b = 1$ the reduction will submit a guess $b' = 1$ with probability $1 - 1/\mathcal{M}$. □

Signature Schemes. Another example of a CKS-compatible game is the signature forgery game. Intuitively, it is clear that adding a testing-oracle that allows the adversary to verify whether they correctly guessed the secret key is of no use — if they could guess the key they could also simply forge a signature.

Lemma 8. *Every signature scheme* $\mathsf{Sig} := (\mathsf{Sig.kg}, \mathsf{Sig.sign}, \mathsf{Sig.vrf})$ *satisfying the standard EUF-CMA security game* $\mathsf{G}_{\mathsf{EUF\text{-}CMA}}$ *notion also satisfies a CKS-compatible one* $\mathsf{G}'_{\mathsf{EUF\text{-}CMA}}$ *for which the challenger works as follows:*

- *The challenger generates a fresh key-pair* $(\mathsf{spk}, \mathsf{ssk})$ *using* Sig.kg *and returns* spk *to the adversary. The secret key* ssk *is output at the private* Keys*.*
- *The* Sign *oracle signs the provided message* m *using the secret key.*
- *The* Expose *returns the secret key and the* Test *oracles checks equality of the secret key.*
- *At the end, when submitting a forgery, it is accepted iff the signature verifies, the message has not been signed before, and the key has not been exposed.*

For any PPT adversary $\mathcal{A}$ *that has non-negligible advantage* $\mathsf{Adv}_{\mathsf{G}'_{\mathsf{EUF\text{-}CMA}}}(\mathcal{A})$*, there exists an appropriately modified PPT adversary* $\mathcal{A}'$ *of roughly the same running time as* $\mathcal{A}$ *that has non-negligible advantage* $\mathsf{Adv}_{\mathsf{G}_{\mathsf{EUF\text{-}CMA}}}(\mathcal{A}')$*.*

Proof. The proof works fairly analogous to the one for the IND-CPA secure encryption one: The essence of the argument is to show that submitting non-trivial valid Test query returning 1, i.e., guessing a secret key, implies winning the basic signature-forgery game. The reduction, that emulates $\mathsf{Adv}_{\mathsf{G}'_{\mathsf{EUF\text{-}CMA}}}(\mathcal{A})$ until such a bad event works as follows: Upon each $\mathsf{Test}(\mathsf{ssk}')$ query, the reduction chooses a fresh message m' (e.g., by sampling a message uniformly at random) then computes $\mathsf{sig}' := \mathsf{Sig.sign}(\mathsf{ssk}', m')$ and checks whether $\mathsf{Sig.vrf}(\mathsf{spk}, m', \mathsf{sig}') = 1$. If so, the reduction outputs (m', sig') as a forgery, and otherwise continues the execution normally. It is easy to see that by correctness of the signature scheme, if $\mathsf{ssk}' = \mathsf{ssk}$, then the check will succeed and (m', sig') be a valid forgery with overwhelming probability. □

One Time Pad for Random Messages. The One Time Pad is not CKS compatible, and indeed deploying our CKS schemes would allow to break privacy in general. When restricted to random messages, however, the One Time Pad is CKS compatible. To this end we consider variant of CKS compatibility for multi-stage games.

Lemma 9. *Let* $\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2)$ *be a two-stage adversary for which* $\mathcal{A}_1$ *outputs two messages of equal length* m_0 *and* m_1*, and a leakage* L*, such that* m_0 *and* m_1 *have high min-entropy given* L*, i.e., is unpredictable given the leakage. The one-time IND-CPA game in which* (m_0, m_1) *is used as a challenge and the ciphertext* c *and* L *is then passed to* $\mathcal{A}_2$ *who then gets access to a* Test *oracle before guessing* b *is single-instance CKS-compatible.*

Proof. Submitting a valid guess k' to the Test oracle implies $\mathcal{A}_2$ guessing m_b: Since $\mathcal{A}_2$ gets $c = m_b \oplus k$, if $k' = k$, we can compute $c \oplus k' = m_b$. Since we assume m_0 and m_1 to be both unpredictable, it thus follows that guessing b (based on c) remains hard. □

B.3 CKS Compatibility of Signal

In this section, we give a high-level overview as to why we believe the Signal protocol (and similar secure messaging protocols) to be CKS compatible. A full-fledged analysis is outside the scope of this work and is left for future work. We use the modularization and security proof by Alwen et al. [2] as a reference.

In general, we propose to apply CKS to the symmetric ratchet layer, as this will yield the best FS/PCS guarantees, for instance allowing to erase individual messages. Alternatively one could also apply CKS to back up the symmetric keys obtained from the asymmetric ratchet layer, saving slightly on cloud storage, bandwidth, and computation. This would mean, however, that only all messages of an asymmetric epoch can be erased at once. Outsourcing the keys of the symmetric ratchet layer comes with a small caveat on how CKS interacts with Signal's immediate decryption property. In principle, CKS assumes that there is one linear sequence of secrets, whereas those keys are indexed by epoch t and message number i within this epoch. However, note that in Signal each message includes the total number of messages that the party sent during their last sending epoch. Therefore, we can nevertheless establish a well-defined linear order (that at times might skip points) without having to worry about additional messages arriving late.

In summary, we can make the secure messaging game (Fig. 2) from [2] CKS compatible as follows:

- **Keys:** Introduce the notion of the symmetric ratchet keys to the security game. In the game by Alwen et al. the double ratchet has been abstracted away for the notion of Secure Messaging. For the concrete instantiation, those keys however still clearly exist. The same has to be done for the FS-AEAD abstraction, i.e. Fig. 5, where those keys are actually used.
- **Testing oracle:** The testing oracle can then allow the adversary to test against those keys, for each given (t, i) pair.
- **Corruption oracle:** Add an explicit corruption oracle for the symmetric ratchet keys. Note that the game obviously can handle those keys leaking, but not at this fine-grained level. Instead, the existing game allows to corrupt parties. Adding fine-grained corruptions should only be a matter of slightly tweaking the safety conditions.

Ignoring the testing oracle for now, it should be relatively straightforward to establish that the Signal protocol is still secure concerning this modified game. One may, therefore, now be tempted to proceed analogous to our example of one-time authenticated encryption and argue that any successful (non-trivial) guess can be turned into an attack against the authenticity property.

Unfortunately, one can see that this does not work. The issue is that in the formalization of Secure Messaging security (Fig. 2) it is assumed that once an i-th message for a given epoch t has been delivered, then all future injection attempts for (i, t) would be automatically rejected. This is a valid assumption, in particular as in the formalization the receiver would immediately delete the respective secret for the sake of FS. This does not imply that Signal is not CKS compatible, but a deficiency in our (naive) proof attempt. There are two potential solutions. Either one could try to establish the futility of the testing oracles via a different, cleverer, reduction. Indeed it intuitively seems implausible as to why an adversary should suddenly be able to guess a secret that has long been erased. It appears that either the adversary could already have guessed the key before it was deleted, or will never be able to do so. Formally establishing this property, however, appears challenging. In particular, a reduction cannot simply "delay" the delivery until the guess was made, to keep the injection valid, as the guess can depend on the party's state later in the protocol and not delivering messages may alter that state.

Alternatively, we propose to instead establish that for the Signal protocol repeated deliveries for the same epoch t and message i are prevented by the protocol, without having to rely on the key

being deleted or the repetition explicitly detected. (That is, we propose that the Signal protocol as is satisfies a stronger definition for which adding the testing oracle becomes easier.) To that end, we would strengthen FS-AEAD and the SM security game to allow breaking authenticity via repeated injection attempts. (Note that the state-corruption oracle should still treat those keys as deleted for the sake of FS.) More specifically, in the FS-AEAD game, the **Inject-B** oracle should allow to rewind the receiver's state. Indeed, one can observe that the FS-AEAD scheme would satisfy this enhanced game — authenticity simply follows from the authenticity of its underlying AE scheme, which allows for multiple injection attempts, and not from the erasure of keys. To quote from the proof of Theorem 5, which establishes FS-AEAD security:

> *In the third hybrid experiment H_3, all AEAD ciphertexts are replaced by random ciphertexts and any (uncompromised) injections are always rejected. Since in H_2 all keys used for the AEAD scheme are random, the indistinguishability of H_3 from H_2 follows immediately from the security of the AEAD scheme.*

With a bit of work, one could therefore formally establish Signal's security with respect to this modified game. It now remains to show that we can add the key-testing oracle without harm. To this end, we emulate the testing oracle as follows:

- Upon input guess k' for epoch and message (t, i), if the symmetric key $k_{t,i}$ for epoch t and message i has been leaked as part of a corruption, then return $k' = k_{t,i}$.
- Otherwise the testing oracle returns 0.

The first case clearly emulates the proper testing oracle faithfully. Hence, any adversary $\mathcal{A}$ who can distinguish the game with the proper testing oracle from the one with the simulated one, must correctly guess a key that has not been leaked. Showing that any such adversary $\mathcal{A}$ can be turned into an adversary against Signal's authenticity follows analogously to the one-time authenticated encryption example from the main body.

C Details on CKS Security

In this section, we describe the CKS-enhanced game $\mathsf{G}_{\mathsf{CKS}}$, presented in Fig. 5, in more detail. Note that Fig. 5 (implicitly) formalizes the challenger $\mathcal{C}_{CKS}$ by describing its oracles, with which the adversary can interact. Furthermore, note that the advantage function α of $\mathsf{G}_{\mathsf{CKS}}$ is defined to be the same as the one of G.

Underlying challenger. The adversary $\mathcal{A}$ can interact with the underlying challenger $\mathcal{C}$ once. Recall that, as this is an interactive process, the adversary can interleave this interaction with calls to other oracles whenever control is handed to the adversary. At the end of this interaction, $\mathcal{C}$ sets the output b, which is also the output of the CKS-enhanced game. (In case $\mathcal{A}$ chooses not to interact with $\mathcal{C}$ or terminates before the interaction ended, the output defaults to $b = 0$.)

CKS interaction. While interacting with the underlying challenger $\mathcal{C}$, the adversary can create CKS users and have them append keys for an epoch e to their state. For this, the adversary can either have the party append the real key from the CKS-compatible game, i.e., as output by $\mathcal{C}.\mathsf{Keys}(\mathsf{e})$, or a key chosen by the adversary. The game keeps track which real keys the user knows as part of the ActualKey mapping. Note that while the game does expose a Retrieve oracle, this does not return a key to the adversary; any use of the key is to be contained in the underlying challenger. Instead, this purely formalizes that the interaction does not reveal information about the keys to the server.

Corruptions. Those keys are then marked as exposed in the underlying challenger $\mathcal{C}$ that can then adjust its behavior — and potentially winning condition — accordingly.

Key delegation. The key delegation process from user u to user u′ is split into two oracles: Grant and Accept. For the former oracle, there is a leak flag indicating whether the message msg is sent over

an secure (leak = $\texttt{false}$) or an insecure (leak = $\texttt{true}$) communication channel. Hence, if leak is set to true, the adversary learns msg. Furthermore, it is assumed that this message leaks all the to be delegated keys and thus all the respective keys are marked as exposed with the underlying challenger $\mathcal{C}$. (As a consequence, leak = $\texttt{true}$ is also only allowed whenever $\mathcal{C}$ permits the exposure of those keys.) In case of leak = $\texttt{false}$, the adversary is given a handle h (which is drawn uniformly at random) to the message msg instead that $\mathcal{A}$ can later use to deliver the message to 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 from $\mathcal{C}$ (as opposed to keys injected by the adversary). In the latter case, of an injected granting message, the game cannot readily determine which keys are from $\mathcal{C}$ — instead it conservatively assumes none to be. (If the game erroneously assumes that a key is injected this only makes $\mathsf{G}_{\mathsf{CKS}}$ easier to win, as corruptions might reveal keys without informing the underlying challenger $\mathcal{C}$. If, conversely, however the game would assume an injected key to be a real one, the game can become needlessly hard to win, as $\mathcal{C}$ might deem certain benign actions to be erroneously “trivializing” the game. Since we want to argue that $\mathsf{G}_{\mathsf{CKS}}$ is not easier to win than G, we thus must err in the first manner.)

FS and PCS. To formalize forward secrecy, the game further allows CKS users to erase certain keys. After such an erasure, corruption of said user must not aid the adversary in winning the underlying game and, thus, do not mark those epochs as corrupted anymore. Further, the game also encodes post-compromise security, as prior corruptions must not influence the security of keys later learned via either Append or Accept.

The random oracle. Finally, let us remark on the use of the ROM. In the above definition, we assume the existence of a fresh random oracle for the CKS scheme. That is, the underlying challenger $\mathcal{C}$ is assumed to be *independent* of the ROM. This allows us circumvent the aforementioned impossibility result.

D Convergent Encryption

Our constructions make use of Convergent Encryption (CE) as introduced by Douceur et al. [27]. The CE scheme makes use of a one-time secure *deterministic* symmetric encryption scheme SE := (SE.Enc, SE.Dec) and a hash function H. Following the abstraction of Bellare, Keelveedhi, and Ristenpart [10] we define CE as a tuple of *deterministic* algorithms $\mathsf{CE} = (\mathsf{CE}.\mathrm{Kg}, \mathsf{CE}.\mathrm{Enc}, \mathsf{CE}.\mathrm{Dec}, \mathsf{CE}.\mathrm{Tag})$:

- ▶ $K \leftarrow \mathsf{CE}.\mathrm{Kg}(m) \coloneqq \mathrm{H}(m)$ computes a key K based on the message m;
- ▶ $C \leftarrow \mathsf{CE}.\mathrm{Enc}(K, m) \coloneqq \mathrm{SE.Enc}(K, m)$ encrypts the message under the aforementioned (message-derived) key;
- ▶ $m \leftarrow \mathsf{CE}.\mathrm{Dec}(K, C) \coloneqq \mathrm{SE.Dec}(K, C)$ decrypts the message;
- ▶ $T \leftarrow \mathsf{CE}.\mathrm{Tag}(C, \mathrm{ad}) \coloneqq \mathrm{H}(C \| \mathrm{ad})$ generates an authentication tag for the ciphertext and some associated data.[19]

Remark 1. Note that while we use the syntax of Message Locked Encryption (MLE) [10] the accompanying security definition is insufficient for our application. Simply put, [10] enforces messages to be sampled independent of the hash function, while our use of CE will not only recursively encrypt ciphertexts, but also let the adversary set part of the message. While Abadi et al. [1] did generalize MLE to lock-dependent messages, their notion, conversely, assumes a general class of lock-dependent distributions. Unfortunately, the CE scheme does not satisfy their notion, while their proposed scheme has to compromise on the integrity notion, clashing with our requirements. We stress that the type of message distribution required for our work lies somewhere in between the two aforementioned notions, composed of a part that is uniform and an adversarially chosen component.

[19] Note that the abstraction of [10] did not include associated data for $\mathsf{CE}.\mathrm{Tag}$.

Correctness and security. For correctness, we require that

$$\mathsf{CE}.\mathrm{Dec}\Big(\mathsf{CE}.\mathrm{Kg}(m),\ \mathsf{CE}.\mathrm{Enc}\big(\mathsf{CE}.\mathrm{Kg}(m),\ m\big)\Big) = m,$$

for any message m, which is implied by correctness of SE. (Recall that all algorithms are deterministic.) In terms of security, we require the following properties from $\mathsf{CE} = (\mathsf{CE}.\mathrm{Kg}, \mathsf{CE}.\mathrm{Enc}, \mathsf{CE}.\mathrm{Dec}, \mathsf{CE}.\mathrm{Tag})$:

1. H will be modeled as a random oracle and, thus, is in particular collision resistant. This implies that $\mathsf{CE}.\mathrm{Tag}$ is collision resistant, and satisfies strong tag consistency as introduced in [10] for the MLE abstraction.
2. The underlying encryption scheme (SE.Enc, SE.Dec) must be one-time IND-CPA secure. That is, based on a single challenge, of two identical-length messages, an adversary cannot guess the challenge bit. In contrast to regular IND-CPA security, no encryption oracle is given.

   Definition 10 (One-time IND-CPA security). *A symmetric encryption scheme* SE = (SE.Enc, SE.Dec) *is* one-time IND-CPA *secure, if*

$$\left| \Pr\left[b = b' \;\middle|\; \begin{array}{l} b \leftarrow\!\!\$\ \{0,1\}; K \leftarrow\!\!\$\ \{0,1\}^\kappa \\ (\mathrm{st}_\mathcal{A}, m_0, m_1) \leftarrow \mathcal{A}(1^\kappa) \\ C \leftarrow \mathrm{SE.Enc}(K, m_b) \\ b' \leftarrow \mathcal{A}(\mathrm{st}_\mathcal{A}, C) \end{array} \right] - \frac{1}{2} \right| \leq \mathsf{negl}(\kappa),$$

   for any PPT adversary $\mathcal{A}$ satisfying $|m_0| = |m_1|$.

3. The convergent encryption scheme must be *non-committing*, to which end we assume the underlying symmetric encryption scheme to be non-committing.

   Definition 11 (Non-committing encryption). *A symmetric encryption scheme* SE = (SE.Enc, SE.Dec) *is* non-committing*, if there exist two efficient algorithms*
   - $C \leftarrow \mathrm{SE.Sim}_1(1^\kappa, |m|)$ *computes a fake ciphertext K based on the security parameter and the message length $|m|$;*
   - $K \leftarrow \mathrm{SE.Sim}_2(C, m)$ *opens a fake ciphertext to a given message m by outputting a corresponding key;*

   such that they produce a ciphertext and key that are indistinguishable from a regular encryption under a uniform random key. That is,

$$\Big| \Pr\big[\mathcal{A}(K, C) \Rightarrow 1 \;\big|\; C \leftarrow \mathrm{SE.Sim}_1(1^\kappa, |m|); K \leftarrow \mathrm{SE.Sim}_2(C, m)\big] - \Pr\big[\mathcal{A}(K, C) \Rightarrow 1 \;\big|\; K \leftarrow\!\!\$\ \{0,1\}^\kappa; C \leftarrow \mathrm{SE.Enc}(K, m)\big] \Big| \leq \mathsf{negl}(\kappa)$$

   for any PPT adversary $\mathcal{A}$.

   Observe that this implies that the CE scheme is non-committing in the programmable random oracle model (where $\mathrm{H}(m)$ is set to be the key output by $\mathrm{SE.Sim}_2$ upon opening).

We remark that while non-committing encryption is generally a hard problem, one-time secure non-committing encryption is satisfied by, for instance, the one-time pad for messages of equal length to the key. Moreover, in the ROM this can easily be extended to support arbitrarily long messages.

E Details on the Line Scheme

In this section, we provide further details on the $\mathsf{CKS_{HA}}$ scheme for the following set of grant, retrieval, and erasure predicates:

$$\begin{aligned}\mathcal{G}_{\mathrm{HA}}(\mathrm{know}, \mathrm{share}) &\coloneqq \mathrm{share} \subseteq \mathrm{know} \wedge \exists i \in \mathbb{N} : \mathrm{share} = \{1, \ldots, i\},\\ \mathcal{R}_{\mathrm{HA}}(\mathrm{know}, \mathrm{share}) &\coloneqq \mathrm{share} \subseteq \mathrm{know} \wedge \exists i, j \in \mathbb{N} : \mathrm{share} = \{i, \ldots, j\},\\ \mathcal{E}_{\mathrm{HA}}(\mathrm{know}, \mathrm{share}) &\coloneqq \mathtt{false},\end{aligned}$$

where the first input refers to the set of epochs for which the party knows the secret — i.e., has learned either through Append or Accept — whereas the latter input refers to the set of secrets the party want to operate the corresponding operation on.

E.1 A Formal Description of the Scheme

A formal description of the line scheme is presented in Fig. 8, with the appropriate helpers already presented in Fig. 9. Note that we use the abstract syntax for convergent encryption as introduced in Appendix D. For simplicity, the description does not include Erase algorithms since the scheme does not allow for erasure, i.e., $\mathcal{E}$ always returns false. Moreover, since $\mathcal{G}$ and $\mathcal{R}$ enforce that for Grant and Retrieve the respective shares of secrets recovered is always an interval, the respective methods take an interval $[i, j]$ as inputs directly.

E.2 Security

Theorem 2. *The line scheme* $\mathsf{CKS_{HA}}$ *satisfies integrity, i.e., for any PPT adversary* $\mathcal{A}$

$$\Pr\Big[\mathrm{CKS}\text{-}\mathsf{Int}^{\mathcal{A}}_{\mathsf{CKS_{HA}}} \Rightarrow 1\Big] \leq \mathsf{negl}(\kappa),$$

if the hash function H *is collision resistant and the* SE *scheme correct.*

Proof. First, observe that in the integrity game, $\mathsf{Secret}[\mathrm{u}, \mathsf{e}]$ gets changed at most twice: when it is initialized and, thus, $\perp$ is replaced with either a concrete secret s or a placeholder $\mathfrak{K}_p$, and in the latter case the placeholder can later be replaced by a concrete value. (Recall that $\mathsf{CKS_{HA}}$ does not support erasures.) By inspection of the protocol it is thus easy to see that $\mathsf{Secret}[\mathrm{u}, \mathsf{e}] \neq \perp$ iff the protocol "knows" key, i.e., iff $\mathsf{e} \leq \mathsf{St}[\mathrm{u}].\mathsf{e_{max}}$ or $\mathsf{St}[\mathrm{u}].\mathsf{Secret}[\mathsf{e}] \neq \perp$. Further, we observe that the winning conditions in the Append and Grant oracles, as well as the first winning condition in the Accept oracle, are trivially ruled out by corresponding checks in the protocol. It, thus, remains to argue that the second winning condition in Accept as well as the winning condition in Retrieve cannot be triggered.

Next, consider a hybrid experiment $\mathcal{H}_1$ that works like $\mathrm{CKS}\text{-}\mathsf{Int}^{\mathcal{A}}_{\mathsf{CKS_{HA}}}$ but (a) disables the aforementioned trivial winning conditions, and (b) additionally keeps track of a mapping Ctxt with $\mathrm{Ctxt}[\mathrm{u}, \mathsf{e}]$ storing the first (valid) CE ciphertext C_e that user u used for epoch e, i.e., the first ciphertext u either uploaded (in Append or Accept) or downloaded and accepted (in Retrieve or Accept). Since this modification does not change the game's behavior or winning condition, clearly $\mathcal{A}$ has the same advantage.

Second, consider a hybrid experiment $\mathcal{H}_2$ that does not use placeholders. Instead, in Accept, it defines $\mathsf{Secret}[\mathrm{u}', \mathsf{e}]$ to be the key resulting from decrypting $\mathrm{Ctxt}[\mathrm{u}', \mathsf{e}]$ instead. (Note that for $\mathsf{CKS_{HA}}$.Accept to not abort the user must not only have obtained such a ciphertext from the server but also successfully decrypted it. Further, recall that the decryption is a deterministic process.) We now observe that this modification can only make the game easier to win $\mathcal{A}$: if this decrypted key is the one that $\mathcal{H}_1$ would later replace the key with, the behavior is unchanged. If it is however a

Protocol Line-CKS

Initialization

U.Init

```
Secret[·] ← ⊥
st ← (0, 0^κ, 0^κ, Secret)
return st
```

S.Init

```
st_S[·,·] ← ⊥
return st_S
```

Appending

U.Append

```
Input: (st, e, s, upload)
  parse (e_max, K_e_max, T_e_max, Secret) ← st
  req e_max < e ∧ Secret[e] = ⊥
  Secret[e] ← s
  st ← (e_max, K_e_max, T_e_max, Secret)
  (st, st_up) ← *compact(st)
  if ¬upload then st_up ← ⊥
  return (st, st_up)
```

U.Upload

```
Input: st_up
  send st_up to S
```

S.Upload

```
Input: st_S
  receive st_up from U
  for all (e, C, T') ∈ st_up do
      T ← CE.Tag(C, T')
      st_S[e, T] ← (C, T')
  return st_S
```

Key retrieval

U.Retrieve

```
Input: (st, (u, v))
  if e_max < v then
      secrets[·] ← ⊥
      for e = u, ..., v do
          req Secret[e] ≠ ⊥
          secrets[e] ← Secret[e]
  else
      send (e_max, T_e_max, u) to S
      receive M_S from S
      secrets ← *recover(st, u, v, M_S)
  return secrets.
```

Key retrieval cont.

S.Retrieve

```
Input: st_S
  receive M_U from U
  parse (e_max, T_e_max, u) ← M_U
  M_S[·] ← ⊥
  for e = e_max, e_max − 1, ..., u do
      (C_e, T_{e−1}) ← st_S[e, T_e]
      M_S[e] ← (C_e, T_{e−1})
  send M_S to U
```

Delegation

U.Grant

```
Input: (st, i)
  parse (e_max, K_e_max, T_e_max, Secret) ← st
  req i ≤ e_max
  send (e_max, T_e_max, i) to S
  receive M_S from S
  for e = e_max, e_max − 1, ..., i + 1 do
      try (·, K_{e−1}, T_{e−1}) ← *invert(K_e, T_e, M_S)
  msg ← (i, K_i, T_i)
  return msg
```

S.Grant

```
Input: st_S
  execute S.Retrieve(st_S)
```

U.Accept

```
Input: (st, i, msg, upload)
  parse (e_max, K_e_max, T_e_max, Secret) ← st
  parse (i', K_i, T_i) ← msg
  req i' = i
  if e_max ≠ 0 then
      z ← e_max
  else if ∃e ≤ i : Secret[e] ≠ ⊥ then
      z ← min{e | Secret[e] ≠ ⊥}
  else
      z ← ⊥
  if z ≠ ⊥ then
      send (i, T_i, z) to S
      receive M_S from S
      st' ← (i, K_i, T_i, Secret)
      secrets' ← *recover(st', z, i, M_S)
      for e = i, i − 1, ..., z do
          if Secret[e] ≠ ⊥ then
              req Secret[e] = secrets'[e]
              Secret[e] ← ⊥
      if e_max ≠ 0 then
          parse (·, T'_e_max) ← M_S[e_max + 1]
          req T'_e_max = T_e_max ∧ secrets'(e_max) = K_e_max
  st ← (i, K_i, T_i, Secret)
  (st, st_up) ← *compact(st)
  if ¬upload then st_up ← ⊥
  return (st, st_up)
```

S.Accept

```
Input: st_S
  execute S.Retrieve(st_S)
```

Fig. 8: The $\mathsf{CKS_{HA}}$ scheme. The corresponding helper methods have been defined in Fig. 9.

Protocol Line-CKS Helpers

***derive**

Input: (s_e, K_{e-1}, T_{e-1})
$K_e \leftarrow \mathsf{CE}.\mathrm{Kg}(s_e \| K_{e-1})$
$C_e \leftarrow \mathsf{CE}.\mathrm{Enc}(K_e, (s_e \| K_{e-1}))$
$T_e \leftarrow \mathsf{CE}.\mathrm{Tag}(C_e, T_{e-1})$
return (C_e, K_e, T_e)

***compact**

Input: st
parse $(\mathsf{e}_{\max}, K_{\mathsf{e}_{\max}}, T_{\mathsf{e}_{\max}}, \mathsf{Secret}) \leftarrow \mathrm{st}$
$\mathrm{st}_{\mathrm{up}} \leftarrow \varnothing$
while $\mathsf{Secret}[\mathsf{e}_{\max}+1] \neq \bot$ **do**
$\mathsf{e}_{\max} \leftarrow \mathsf{e}_{\max}+1$
$(C_{\mathsf{e}_{\max}}, K_{\mathsf{e}_{\max}}, T_{\mathsf{e}_{\max}}) \leftarrow \texttt{*derive}(\mathsf{Secret}[\mathsf{e}_{\max}], K_{\mathsf{e}_{\max}-1}, T_{\mathsf{e}_{\max}-1})$
$\mathsf{Secret}[\mathsf{e}_{\max}] \leftarrow \bot$
$\mathrm{st}_{\mathrm{up}} \leftarrow \mathrm{st}_{\mathrm{up}} \cup \{(\mathsf{e}_{\max}, C_{\mathsf{e}_{\max}}, T_{\mathsf{e}_{\max}-1})\}$
$\mathrm{st} \leftarrow (\mathsf{e}_{\max}, K_{\mathsf{e}_{\max}}, T_{\mathsf{e}_{\max}}, \mathsf{Secret})$
return $(\mathrm{st}, \mathrm{st}_{\mathrm{up}})$

***invert**

Input: (K_e, T_e, M_S)
parse $(C_e, T_{e-1}) \leftarrow M_S[\mathsf{e}]$
req $T_e = \mathsf{CE}.\mathrm{Tag}(C_e, T_{e-1})$
$m \leftarrow \mathsf{CE}.\mathrm{Dec}(K_e, C_e)$
parse $m \leftarrow (s_e, K_{e-1})$
return (s, K_{e-1}, T_{e-1})

***recover**

Input: (st, i, j, M_S)
parse $(\mathsf{e}_{\max}, K_{\mathsf{e}_{\max}}, T_{\mathsf{e}_{\max}}, \mathsf{Secret}) \leftarrow \mathrm{st}$
$\mathrm{secrets}[\cdot] \leftarrow \bot$
for $\mathsf{e} = \mathsf{e}_{\max}, \mathsf{e}_{\max}-1, \ldots, i$ **do**
try $(s_e, K_{e-1}, T_{e-1}) \leftarrow \texttt{*invert}(K_e, T_e, M_S)$
if $\mathsf{e} \leq j$ **then** $\mathrm{secrets}[\mathsf{e}] \leftarrow s_e$
return secrets

Fig. 9: The core methods for the $\mathsf{CKS}_{\mathsf{HA}}$ scheme.

different one, this now means that where $\mathcal{H}_1$ would substitute the placeholder $\mathcal{H}_2$ will be won directly before, e.g., failing the $\mathsf{Secret}[\mathrm{u}, \mathsf{e}] \neq \mathrm{secrets}(\mathsf{e})$ check in the Retrieve oracle.

Third, consider a hybrid $\mathcal{H}_3$ that upon Accept with an honest grant checks that the accepting user u' and the granting user U agree on the history of CE ciphertexts. That is, the adversary wins if $\mathrm{Ctxt}[\mathrm{u}, \mathsf{e}] \neq \mathrm{Ctxt}[\mathrm{u}', \mathsf{e}]$ for some epoch $\mathsf{e} \in \mathrm{share} = [i, j]$. Clearly, if $\mathsf{CKS}_{\mathsf{HA}}$.Accept processes the input, then they agree on T_j. Since U' validates the hash chain on the CE ciphertexts and tags, collision resistance of H thus implies that this check only fails with negligible probability.

Fourth, consider a hybrid $\mathcal{H}_4$ that additionally enforces the invariant that, if set, the key resulting from decrypting $\mathrm{Ctxt}[\mathrm{u}, \mathsf{e}]$ is equal to $\mathsf{Secret}[\mathrm{u}, \mathsf{e}]$, with the adversary winning otherwise. If $\mathrm{Ctxt}[\mathrm{u}, \mathsf{e}]$ is set upon the user uploading the ciphertext, then the invariant is maintained due to correctness of the CE scheme. If $\mathrm{Ctxt}[\mathrm{u}', \mathsf{e}]$ is set as part of Accept, there are two options. First, if $\mathsf{Secret}[\mathrm{u}', \mathsf{e}]$ is already set, but $\mathrm{Ctxt}[\mathrm{u}', \mathsf{e}]$ not, then Append must have been called for an out-of-order epoch, at which point $\mathsf{Secret}[\mathrm{u}', \mathsf{e}] = \mathsf{St}[\mathrm{u}'].\mathsf{Secret}[\mathsf{e}]$ was trivially satisfied. Since the protocol checks that the new $\mathrm{Ctxt}[\mathrm{u}', \mathsf{e}]$ decrypts to $\mathsf{St}[\mathrm{u}'].\mathsf{Secret}[\mathsf{e}]$ the invariant is hence preserved. Second, if $\mathsf{Secret}[\mathrm{u}', \mathsf{e}]$ gets assigned $\mathsf{Secret}[\mathrm{u}, \mathsf{e}]$, then we know from the invariant introduced in $\mathcal{H}_3$ that the parties have the same ciphertext. Hence, the invariant still holding for the granting party u implies that the invariant is preserved for u' as well.

Finally, we argue that triggering either of the two remaining initial winning conditions in $\mathcal{H}_4$ happens with at most negligible probability. First, consider the check $\mathsf{Secret}[\mathrm{u}', \mathsf{e}] \in \{\bot, \mathrm{s}\}$ failing, where s is the value from $\mathsf{Secret}[\mathrm{u}, \mathsf{e}]$. By our invariants we have however that (1) $\mathrm{Ctxt}[\mathrm{u}, \mathsf{e}] = \mathrm{Ctxt}[\mathrm{u}', \mathsf{e}]$ and (2) for both parties the decryption of their respective ciphertexts equals to the key stored in Secret. Second, consider the check $\mathsf{Secret}[\mathrm{u}, \mathsf{e}] = \mathrm{secrets}(\mathsf{e})$ in Retrieve being broken. If $\mathrm{secrets}(\mathsf{e})$ is set by the protocol as $\mathsf{St}[\mathrm{u}].\mathsf{Secret}[\mathsf{e}]$, then the check trivially holds, as argued before. If, on the other hand, the protocol derives the key by retrieving the respective ciphertext and decrypt it, note that collision resistance implies that the protocol only accepts the same ciphertext as stored in $\mathrm{Ctxt}[\mathrm{u}, \mathsf{e}]$. Hence, our invariant from $\mathcal{H}_4$ implies the check to not fail either. □

Theorem 3. *The line scheme* $\mathsf{CKS}_{\mathsf{HA}}$ *is secure, i.e., for any CKS-compatible game* G *and any admissible PPT adversary* $\mathcal{A}$ *there exists a PPT adversary* $\mathcal{A}'$ *such that*

$$\mathsf{Adv}_{\mathsf{G}_{\mathsf{CKS}}}(\mathcal{A}) \leq \mathsf{Adv}_{\mathsf{G}}(\mathcal{A}') + \mathsf{negl}(\kappa),$$

if the SE *scheme is correct, non-committing, and one-time IND-CPA secure, and the hash function is modeled as a random oracle.*

Proof. Fix some arbitrary CKS-compatible game G, and an arbitrary PPT adversary $\mathcal{A}$ against $\mathsf{G}_{\mathsf{CKS}}$. In the following we construct an adversary $\mathcal{A}'$ against G that has almost the same advantage as $\mathcal{A}$ against their respective games.

$\mathcal{A}'$ internally runs $\mathcal{A}$ and forwards its interaction with the challenger $\mathcal{C}$. In addition, $\mathcal{A}'$ emulates the various CKS oracles towards $\mathcal{A}$. To this end, $\mathcal{A}'$ keeps track of the various variables such as n and ActualKey$[\cdot]$ of $\mathsf{G}_{\mathsf{CKS}}$, except for $\mathsf{St}[\cdot]$. Instead, it keeps track of the following additional state:

- $\mathsf{Secret}[\mathsf{e}]$ stores the e-th secret from the CKS-compatible game G, once known to the reduction.
- $\mathrm{Secret}[\mathrm{u}, \mathsf{e}]$ stores the secret user u used in epochs e. If u uses a key from the CKS-compatible game G that is not known to the reduction yet, it uses $\mathfrak{K}_{\mathsf{e}}$ as a placeholder instead.
- $T[\mathrm{u}, \mathsf{e}]$ stores the tag the user u used in epochs e.
- $K[\mathrm{u}, \mathsf{e}]$ stores the respective CE key, if known.
- $C[T_{\mathsf{e}-1}, \mathrm{s}_{\mathsf{e}}]$ stores the CE ciphertexts generated when encrypting a user previously having tag $T_{\mathsf{e}-1}$ appends s_{e}. Here, s_{e} can be a placeholder.

Using that state, $\mathcal{A}'$ emulates the CKS oracles of $\mathsf{G}_{\mathsf{CKS}}$ as follows:

- **CreateUser:** Emulates the CKS protocol by setting up an initial state $(\mathsf{e}_{\mathsf{max}}, K_{\mathsf{e}_{\mathsf{max}}}, T_{\mathsf{e}_{\mathsf{max}}}, \mathsf{Secret}) = (0, 0^k, 0^\ell, \mathsf{Secret})$ for an empty mapping Secret.
- **Corrupt:** Emulates the $\mathsf{G}_{\mathsf{CKS}}$ oracle by exposing all the epochs the corrupted user knows, learning the actual keys. For each new key s_{e} it learns, it
  - sets $\mathsf{Secret}[\mathsf{e}] \leftarrow \mathrm{s}_{\mathsf{e}}$
  - replaces $\mathfrak{K}_{\mathsf{e}}$ with s_{e} in $\mathrm{Secret}[\mathrm{u}', \mathsf{e}']$ for all u' and e'.
  - copies $C[T'_{\mathsf{e}-1}, \mathfrak{K}_{\mathsf{e}}]$ over to $C[T'_{\mathsf{e}-1}, \mathrm{s}_{\mathsf{e}}]$ for each $T'_{\mathsf{e}-1}$ such that the former was defined.

  For each user u' and epoch e, let $\mathrm{s}_1, \dots, \mathrm{s}_{\mathsf{e}} = \mathrm{Secret}[\mathrm{u}', 1], \dots, \mathrm{Secret}[\mathrm{u}', \mathsf{e}]$ and do the following. If s_{e} is not a placeholder and $K[\mathrm{u}', \mathsf{e}-1]$ is set, but $K[\mathrm{u}', \mathsf{e}]$ has not been set, then we now "open" the fake ciphertext $C_{\mathrm{u}', \mathsf{e}} \coloneqq C[T[\mathrm{u}', \mathsf{e}-1], \mathrm{s}_{\mathsf{e}}]$. That is, the adversary $\mathcal{A}'$:
  - Sets $K[\mathrm{u}', \mathsf{e}] \leftarrow \mathrm{SE.Sim}_2(C_{\mathrm{u}, \mathsf{e}}, (\mathrm{s}_{\mathsf{e}} \| K[\mathrm{u}, \mathsf{e}-1]))$
  - Copies the key to every other user u'' that uses the same tag and s_{e}.
  - Programs the ROM s.t. $\mathsf{RO}[\mathrm{s}_{\mathsf{e}} \| K_{\mathrm{u}', \mathsf{e}-1}] = K_{\mathrm{u}', \mathsf{e}}$

  Finally, $\mathcal{A}'$ assembles a CKS protocol state as follows:
  - computes $\mathsf{e}_{\mathsf{max}}$ as the smallest e such that $\mathsf{Known}[\mathrm{u}, \mathsf{e}+1] = \mathtt{false}$
  - sets $K_{\mathsf{e}_{\mathsf{max}}} \leftarrow K[T[\mathrm{u}, \mathsf{e}_{\mathsf{max}}-1], \mathrm{Secret}[\mathrm{u}, \mathsf{e}_{\mathsf{max}}]]$ and $T_{\mathsf{e}_{\mathsf{max}}} \leftarrow T[\mathrm{u}, \mathsf{e}_{\mathsf{max}}]$, respectively.
  - for all $\mathsf{e} > \mathsf{e}_{\mathsf{max}}$ such that Known, sets $\mathsf{Secret}[\mathsf{e}] \leftarrow \mathrm{Secret}[\mathrm{u}, \mathsf{e}]$
- **Append:** $\mathcal{A}$ rejects if the user u already knows a key for the given epoch, i.e., if $\mathsf{Known}[\mathrm{u}, \mathsf{e}] \neq \perp$. If $\mathrm{s} = \perp$, then set $\mathrm{s} \leftarrow \mathsf{Secret}[\mathsf{e}]$ in case that is defined, and $\mathfrak{K}_{\mathsf{e}}$, otherwise. Set $\mathrm{Secret}[\mathrm{u}, \mathsf{e}] \leftarrow \mathrm{s}$. If $C[T[\mathrm{u}, \mathsf{e}-1], \mathrm{s}]$ is already defined, then find another user U' such that $T[\mathrm{u}, \mathsf{e}-1] = T[\mathrm{u}', \mathsf{e}-1]$ and $\mathrm{s} = \mathrm{Secret}[\mathrm{u}', \mathsf{e}]$ and set
  - $T[\mathrm{u}, \mathsf{e}] \leftarrow T[\mathrm{u}', \mathsf{e}]$.

  Afterwards, define $\mathrm{st}_{\mathrm{up}} \coloneqq (\mathsf{e}, C[T[\mathrm{u}, \mathsf{e}-1], \mathrm{s}], T[\mathrm{u}, \mathsf{e}-1])$ and hand that to $\mathcal{A}$ in case upload $= \mathtt{true}$. Else, generate the ciphertext and tag as follows:
  - If s is not a placeholder and $K[\mathrm{u}, \mathsf{e}-1]$ has been set before, then generate a regular ciphertext. That is, set

$$
\begin{aligned}
K[\mathrm{u}, \mathsf{e}-1] &\leftarrow \mathsf{RO}[\mathrm{Secret}[\mathrm{u}, \mathsf{e}] \| K_{\mathrm{u}, \mathsf{e}-1}] \\
C[T[\mathrm{u}, \mathsf{e}-1], \mathrm{s}] &\leftarrow \mathsf{CE}.\mathrm{Enc}(K_{\mathrm{u}, \mathsf{e}}, (\mathrm{s} \| K_{\mathrm{u}, \mathsf{e}-1})) \\
&= \mathrm{SE.Enc}(K_{\mathrm{u}, \mathsf{e}}, (\mathrm{s} \| K_{\mathrm{u}, \mathsf{e}-1}))
\end{aligned}
$$

    sampling the random oracle if necessary. Copy the key to any other user that has the same tag $T[\mathrm{u}, \mathsf{e}-1]$ and secret.
  - Otherwise, set $C[T[\mathrm{u}, \mathsf{e}-1], \mathrm{s}] \leftarrow \mathrm{SE.Sim}_1(1^\kappa, n)$, where n denotes the combined length of the appended key, a CE key, and a CE tag.

- Compute $T[\mathrm{u},\mathsf{e}] \leftarrow \mathsf{CE}.\mathrm{Tag}(C_{\mathrm{u},\mathsf{e}}, T[\mathrm{u},\mathsf{e}-1]) := \mathsf{RO}[C_{\mathrm{u},\mathsf{e}} \| T[\mathrm{u},\mathsf{e}-1]]$, sampling a uniform random value for RO if necessary. Copy the key to any other user that has the same tag $T[\mathrm{u},\mathsf{e}-1]$ and secret.

 For any further consecutively known key (e.g., if $\mathsf{Known}[\mathrm{u},\mathsf{e}+1] \neq \perp$) repeat the above steps to outsource the key.

– **Grant:** For granting access to epochs share $= [1,i]$, $\mathcal{A}'$ first emulates the run of the protocol by fetching $(\mathsf{e}_{\mathsf{max}}, T_{\mathsf{e}_{\mathsf{max}}}, i)$ to the adversary, receiving $\{(j, C_j)\}_{j=i}^{\mathsf{e}_{\mathsf{max}}}$ in return. To emulate the protocol verifying those ciphertexts, $\mathcal{A}'$ accepts them iff $C_j = C_{\mathrm{u},j}$, i.e., iff the server returns the same ciphertext that user originally uploaded.

 If leak = `false`, $\mathcal{A}'$ then samples a uniform random handle h and records $\mathrm{Msgs}[\mathrm{h}] \leftarrow (\mathrm{share}, \mathrm{U})$. That is, instead of recording the granting message msg, it simply records user and the share. Then it outputs the handle h.

 If leak = `true`, then $\mathcal{A}'$ invokes $\mathcal{C}.\mathsf{Expose}(\mathsf{e})$ for $\mathsf{e} = 1, \ldots, i$ and proceeds analogous to the Corrupt oracle to update K, C, T, and Secret. Finally, it outputs $\mathrm{msg} = (i, K_{\mathrm{u},i}, T_{\mathrm{u},i})$.

– **Accept:** If $\mathcal{A}$ delivers a handle h to u′, then $\mathcal{A}'$ emulates the Accept oracle using a simple consistency check and copying the relevant information. Let U denote the user that shared share $= [i]$, i.e., $\mathrm{Msgs}[\mathrm{h}] = (\mathrm{share}, \mathrm{U})$.
  - $\mathcal{A}'$ rejects the operation if $\mathrm{Secret}[\mathrm{u},\mathsf{e}] \neq \mathrm{Secret}[\mathrm{u}',\mathsf{e}]$, but the latter is set, for any $\mathsf{e} \leq i$.
  - Then copies $\mathrm{Secret}[\mathrm{u}',\mathsf{e}] \leftarrow \mathrm{Secret}[\mathrm{u},\mathsf{e}]$, and analogously for $T[\mathrm{u}',\mathsf{e}] \leftarrow T[\mathrm{u},\mathsf{e}]$.
  - Analogous to the protocol, $\mathcal{A}'$ then also outsources encryptions for epoch $j = i+1, \ldots$ as long as $\mathrm{Secret}[\mathrm{u}', j]$ is set. To this end, $\mathcal{A}'$ either generates real or fake ciphertexts as described in the Append oracle.

 If $\mathcal{A}$ injects a grant message $\mathrm{msg} = (i, K_i, T_i)$, the reduction emulates the performing the consistency checks, if required. That is, it fetches the ciphertexts that allow to decrypt up to epoch z, where z denotes the smallest epoch for which the party so far knew the secret. If those checks succeed, it sets $T[\mathrm{U},\mathsf{e}]$ and $\mathrm{Secret}[\mathrm{u},\mathsf{e}]$ for $z \leq \mathsf{e} \leq i$. It also populates $C[T_{\mathsf{e}-1}, \mathrm{s}_{\mathsf{e}}]$ for all tags and secrets it learns during that interaction. In case no consistency check is performed, the reduction simply stores $T[\mathrm{U},\mathsf{e}]$.

– **Retrieve:** The reduction simply simulates the message sent by the corresponding party to the server. Note that this message contains public information only. If by emulating the protocol, $\mathcal{A}'$ learns new ciphertexts, tags, CE keys, or secrets, populate the maintained state accordingly.

– **RO:** Upon input $x \in \{0,1\}^*$, $\mathcal{A}'$ returns $\mathsf{RO}[x]$ if that has already been defined. Otherwise, it proceeds as follows:
  - Try to parse x as $\mathrm{s}_{\mathsf{e}} \| K_{\mathsf{e}-1}$ (from $\mathsf{CE}.\mathrm{Kg}$). If this succeeds, check whether there exists a user U and a node v such that the query has to be answered consistently and set RO accordingly. More specifically:
    * Check that x can be parse as secret and CE key, for $K_{\mathsf{e}-1}$ to $\mathcal{A}'$.
    * Then use the Test oracle with input s_{e} to see whether s_{e} is indeed the correct secret.
    * If so, $\mathcal{A}'$ ensures everything is consistent by opening C_{e} (using the non-committing property) and programming $\mathsf{RO}[x]$ to return a key that decrypts C_{e} consistently. Afterward, return $\mathsf{RO}[x]$.
  - If $\mathsf{RO}[x]$ was not already defined and did not need programming, sample $\mathsf{RO}[x]$ u.a.r. and return that value.

It remains to argue that this reduction is successful, i.e., the original adversary winning the CKS-enhanced game with non-negligible probability results in the reduction winning the underlying game with non-negligible probability.

The main difference are the simulated ciphertexts: whereas in the CKS-enhanced game the adversary $\mathcal{A}$ always receives real ciphertexts, the reduction hands them simulated ones, if the reduction doesn't know the message to be encrypted. By the IND-CPA security of the encryption scheme, this is indistinguishable, since $\mathcal{A}$ has no information on the encryption key (which is $\mathrm{H}(\mathrm{s}_i \| K_{i-1})$ with the hash modeled as a random oracle). Once $\mathcal{A}$ does learn both s_i and K_{i-1}, e.g., via corrupting a party,

the reduction uses the non-committing property of the CE scheme to generate K_i explaining the ciphertext, and programming $\mathrm{H}(\mathrm{s}_i \| K_{i-1})$ accordingly. ($\mathcal{A}$ only queried the random oracle at this position with negligible probability.)

Whenever a party fetches ciphertexts from the server, the collision resistance of the tags (computed as a hash) ensure that they reject everything but the ciphertexts they originally uploaded. As such, whenever a party grants access to a prefix, the reduction can safely assume that the receiving party ends up with the same keys as long as the adversary delivers a handle.

If the adversary injects a grant message instead, observe that the message — consisting of an epoch number, CE key and tag — determines the entire prefix of tags and ciphertexts, given the tags' collision resistance. Hence, the reduction looking ciphertexts by tag and secret, ensures consistency even if one party gets injected a grant message that later turns out to be consistent with the state reached by an honest party (that may have been compromised). □

F Details on the Interval Scheme

In this section, we now consider the $\mathsf{CKS}_{\mathsf{Interval}}$ scheme for the following set of grant, retrieval, and erasure predicates:

$$\begin{aligned}\mathcal{G}_{\text{Interval}}(\text{know}, \text{share}) = \mathcal{R}_{\text{Interval}}(\text{know}, \text{share}) = \mathcal{E}_{\text{Interval}}(\text{know}, \text{share}) \\ := \text{share} \subseteq \text{know} \land \exists i, j \in \mathbb{N} : \text{share} = \{i, \dots, j\}.\end{aligned}$$

F.1 A Formal Description of the Scheme

A formal description of the line scheme is presented in Fig. 11, with the core helper functions presented in Fig. 10 and additional ones shown in Fig. 12. Moreover, since $\mathcal{G}$, $\mathcal{R}$, and $\mathcal{E}$ enforce that the respective shares of secrets are intervals, the respective methods take an interval $[i, j]$ as inputs directly.

The scheme works for a bounded number of epochs $E = 2^h$. Parties store a binary tree τ of height h, with each node v having associated properties $v.K$ and $v.T$ for a CE key and tag, respectively. (At leaves we abuse $v.K$ to store the epoch's secret instead.) We assume the tree to be compactly represented such that storage grows only in the number of vertices with assigned properties.

Protocol Interval-CKS Helpers

```
*derive

Input: (K_ℓ, T_ℓ, K_r, T_r)
  K ← CE.Kg(K_ℓ‖K_r)
  C ← CE.Enc(K, (K_ℓ‖K_r))
  T ← CE.Tag(C, (T_ℓ‖T_r))
  return (C, K, T)

*compact

Input: v
  // Compacts state after learned key of node v
  st_up ← ∅
  v ← v.parent
  while v ≠ ⊥ do
     v_ℓ ← v.lChild
     v_r ← v.rChild
     if v_ℓ.K ≠ ⊥ ∧ v_r.K ≠ ⊥ then
        (C, v.K, v.T) ← *derive(v_ℓ.K, v_ℓ.T, v_r.K, v_r.T)
        st_up ← st_up ∪ {(v.idx, C, v_ℓ.T, v_r.T)}
        (v_ℓ.K, v_ℓ.T) ← (⊥, ⊥)
        (v_r.K, v_r.T) ← (⊥, ⊥)
        v ← v.parent
     else
        v ← ⊥
  return st_up

*invert

Input: (C, K, T, T_ℓ, T_r)
  req T = CE.Tag(C, (T_ℓ‖T_r))
  parse (K_ℓ, K_r) ← CE.Dec(K, C)
  return (K_ℓ, K_r)

*recover

Input: (v, M)
  // Recover key of node v using server message M
  if v.K ≠ ⊥ then return
  try *recover(v.parent, M)
  parse (C, T_ℓ, K_ℓ) ← M[v.parent.idx]
  try (K_ℓ, K_r) ← *invert(C, v.parent.K, v.parent.T, T_ℓ, T_r)
  (v.parent.lChild.K, v.parent.lChild.T) ← (K_ℓ, T_ℓ)
  (v.parent.rChild.K, v.parent.rChild.T) ← (K_r, T_r)

*cover

Input: (v, idx_i, idx_j)
  if idx_i ≤ v.minLeafIdx ∧ idx_j ≥ v.maxLeafIdx then
     return {v}
  else if idx_i > v.minLeafIdx ∨ idx_j < v.maxLeafIdx then return ∅
  else return *cover(v.lChild, idx_i, idx_j)
                 ∪ *cover(v.rChild, idx_i, idx_j)
```

Fig. 10: The core methods for the $\mathsf{CKS}_{\mathsf{Interval}}$ scheme.

We use object-oriented notation such as $v.\mathsf{parent}$, $v.\mathsf{lChild}$, and $v.\mathsf{rChild}$ to denote a node's parent, as well as left and right children, respectively. Each node furthermore has an index $v.\mathsf{idx}$ according to *in-order traversal*. A node can be retrieves via its index using $\tau.\mathsf{nodes}[\mathsf{idx}]$. Leafs are additionally numbered from left to right, with $w = \tau.\mathsf{leaves}[k]$ returning the k-th leaf, and $w.\mathsf{leafNumber} = k$, respectively.

The server for each node maintains a mapping $v.\text{data}$ from tag T to a ciphertext C and tags T_ℓ and T_r of the left and right child nodes.

Protocol Interval-CKS

Initialization

U.Init

```
τ ← new full binary tree of height h, where each
      node v has properties (v.K, v.T)
return τ
```

S.Init

```
τ_S ← new full binary tree of height h, where each
      node v has mapping v.data[T] = (C, T_ℓ, T_r)
return τ_S
```

Appending

U.Append

```
Input: (τ, e, s, upload)
  v ← τ.leaves[e]
  v.K ← s
  v.T ← 0^κ
  st_up ← *compact(v.parent) // Compact upwards
  if ¬upload then st_up ← ⊥
  return (τ, st_up)
```

U.Upload

```
Input: st_up
  send st_up to S
```

S.Upload

```
Input: τ_S
  receive st_up from U
  for all (idx, C, T_l, T_r) ∈ st_up do
     T ← CE.Tag(C, (T_l, T_r))
     v ← τ_S.nodes[idx]
     v.data[T] ← (C, T_l, T_r)
  return τ_S
```

Key retrieval

U.Retrieve

```
Input: (τ, (i, j))
  τ' ← clone(τ)
  (idx_i, idx_j) ← (τ'.leaves[i].idx, τ'.leaves[j].idx)
  M_U ← ∅
  cov ← *cover(τ'.root, idx_i, idx_j)
  for v ∈ cov do
     v' ← *storedAncestor(v)
     M_U ← M_U ∪ {(v.idx, v'.idx, v'.T)}
  send M_U to S
  receive M_S from S
  secrets[·] ← ⊥
  for idx ∈ idx_i, ..., idx_j do
     v ← τ'.nodes[idx]
     if v.isleaf then
        try *recover(v, M_S)
        secrets[v.leafNumber] ← v.K
  return secrets
```

Key retrieval contd.

S.Retrieve

```
Input: τ_S
  receive M_U from U
  M_S[·] ← ⊥
  for (idx, idx', T') ∈ M_U do
     v ← τ_S.nodes[idx]; v' ← τ_S.nodes[idx']
     M_S ← M_S ∪ *servePath(v, v', T')
     M_S ← M_S ∪ *serveTree(v, M_S[v.idx])
  send M_S to U
```

Erasure

U.Erase

```
Input: (τ, (i, j))
  (idx_i, idx_j) ← (τ.leaves[i].idx, τ.leaves[j].idx)
  safe ← *safeResidual(τ.root, idx_i, idx_j)
  M_U ← ∅
  for v ∈ safe do
     v' ← *storedAncestor(v)
     if v' ≠ ⊥ ∧ v' ≠ v then
        M_U ← M_U ∪ {(v.idx, v'.idx, v'.T)}
     else
        safe ← safe \ {v}
  send M_U to S
  receive M_S from S
  for v ∈ safe do
     try *recover(v, M_S)
  for v ∈ safe do
     (v.sibling.K, v.sibling.K) ← (⊥, ⊥)
     while v.parent ≠ ⊥ do
        (v.parent.K, v.parent.K) ← (⊥, ⊥)
        v ← v.parent
  return τ
```

S.Erase

```
Input: τ_S
  execute S.Grant(τ_S)
```

Access delegation

U.Grant

```
Input: (τ, (i, j))
  τ' ← clone(τ)
  (idx_i, idx_j) ← (τ'.leaves[i].idx, τ'.leaves[j].idx)
  msg, M_U ← ∅
  cov ← *cover(τ'.root, idx_i, idx_j)
  for v ∈ cov do
     v' ← *storedAncestor(v)
     M_U ← M_U ∪ {(v.idx, v'.idx, v'.T)}
  send M_U to S
  receive M_S from S
  for v ∈ cov do
     try *recover(v, M_S)
     msg ←∪ {(v.idx, v.K, v.T)}
  return msg
```

Access delegation contd.

S.Grant

Input: τ_S
receive M_U from U
$M_S[\cdot] \leftarrow \bot$
for $(\mathsf{idx}, \mathsf{idx}', T') \in M_U$ **do**
  $v \leftarrow \tau_S.\mathsf{nodes}[\mathsf{idx}]$; $v' \leftarrow \tau_S.\mathsf{nodes}[\mathsf{idx}']$
  try $P \leftarrow$ *servePath(v, v', T')
  $M_S \leftarrow M_S \cup P$
send M_S to U

S.Accept

Input: τ_S
execute S.Grant(τ_S)

U.Accept

Input: $(\tau, (i, j), \mathrm{msg}, \mathrm{upload})$
$M_U \leftarrow \varnothing$
for $(\mathsf{idx}, K, T) \in \mathrm{msg}$ **do**
  $v \leftarrow \tau.\mathsf{nodes}[\mathsf{idx}]$
  for all $v' : v'.\mathsf{descendantOf}(v) \wedge v'.K \neq \bot$ **do**
    $M_U \leftarrow M_U \cup \{(v'.\mathsf{idx}, \mathsf{idx}, T)\}$
send M_U to S
receive M_S from S
$\tau' \leftarrow \mathrm{clone}(\tau)$
for $(\mathsf{idx}, K, T) \in \mathrm{msg}$ **do**
  $v \leftarrow \tau'.\mathsf{nodes}[\mathsf{idx}]$
  if *storedAncestor$(v) \neq \bot$ **then skip**
  $(v.K, v.T) \leftarrow (K, T)$
  for $(\mathsf{idx}', \cdot, \cdot) \in M_U$ **do**
    // check consistency
    $v' \leftarrow \tau'.\mathsf{nodes}[\mathsf{idx}']$
    $(K', T') \leftarrow (v'.K, v'.T)$
    $(v'.K, v'.T) \leftarrow (\bot, \bot)$
    try *recover(v', M_S)
    req $(K', T') = (v'.K, v'.T)$
    // purge from actual tree
    $(\tau.\mathsf{nodes}[\mathsf{idx}'].K, \tau.\mathsf{nodes}[\mathsf{idx}'].T) \leftarrow (\bot, \bot)$
for $(\mathsf{idx}, K, T) \in \mathrm{msg}$ **do**
  $v \leftarrow \tau.\mathsf{nodes}[\mathsf{idx}]$
  if *storedAncestor$(v) \neq \bot$ **then skip**
  $(v.K, v.T) \leftarrow (K, T)$
  $\mathrm{st}_{\mathrm{up}} \leftarrow \mathrm{st}_{\mathrm{up}} \cup$ *compact(v)
if $\neg$upload **then** $\mathrm{st}_{\mathrm{up}} \leftarrow \bot$
return $(\tau, \mathrm{st}_{\mathrm{up}})$

Fig. 11: The $\mathsf{CKS}_{\mathsf{Interval}}$ scheme supporting up to 2^h epochs.

Protocol Interval-CKS Helpers

*storedAncestor

Input: v
// Finds ancestor from which to recover.
if $v = \bot \vee v.K \neq \bot$ **then return** v
else return *storedAncestor$(v.\mathsf{parent})$

*servePath

Input: (v, v', T')
// Serve the path from v' down to v, starting with tag T'
$M[\cdot] \leftarrow \bot$
try $(C, T_\ell, T_r) \leftarrow v'.\mathrm{data}[T]$
$M[v'.\mathsf{idx}] \leftarrow (C, T_\ell, T_r)$
if $v \neq v'$ **then**
  if $v.\mathsf{descendantOf}(v'.\mathsf{lChild})$ **then**
    try $P \leftarrow$ *servePath$(v, v'.\mathsf{lChild}, T_\ell)$
  else
    try $P \leftarrow$ *servePath$(v, v'.\mathsf{rChild}, T_r)$
  $M \leftarrow M \cup P$
return M

*serveTree

Input: $(v, (C, T_\ell, T_r))$
// Serve the subtree rooted at v
$M[\cdot] \leftarrow \bot$
$M[v.\mathsf{idx}] \leftarrow (C, T_\ell, T_r)$
if $\neg v.\mathsf{isleaf}$ **then**
  $M \leftarrow M \cup$ *serveTree$(v.\mathsf{lChild}, v.\mathsf{lChild}.\mathrm{data}[T_\ell])$
    $\cup$ *serveTree$(v.\mathsf{rChild}, v.\mathsf{rChild}.\mathrm{data}[T_r])$
return M

*safeResidual

Input: $(v, \mathsf{idx}_i, \mathsf{idx}_j)$
// A minimal cover for that allows to restore all nodes except for the interval idx_i to idx_j.
if $\mathsf{idx}_j < v.\mathsf{minLeafIdx} \vee \mathsf{idx}_i > v.\mathsf{maxLeafIdx}$ **then**
  return $\{v\}$
else if $\mathsf{idx}_i \leq v.\mathsf{minLeafIdx} \wedge \mathsf{idx}_j \geq v.\mathsf{maxLeafIdx}$ **then**
  return $\varnothing$
else
  return *safeResidual$(v.\mathsf{lChild}, \mathsf{idx}_i, \mathsf{idx}_j)$
    $\cup$ *safeResidual$(v.\mathsf{rChild}, \mathsf{idx}_i, \mathsf{idx}_j)$

Fig. 12: Additional helper methods for the $\mathsf{CKS}_{\mathsf{Interval}}$ scheme.

F.2 Security

Theorem 5. *Assuming the* CE *scheme is correct and one-time IND-CPA secure and modeling* H *as a random oracle, the interval scheme* $\mathsf{CKS}_{\mathsf{Interval}}$ *satisfies integrity and is secure. That is, for any CKS-compatible game* G*, and any admissible PPT adversary* $\mathcal{A}$*, there exists a PPT adversary* $\mathcal{A}'$ *such that*

$$\mathsf{Adv}_{\mathsf{G}_{\mathsf{CKS}}}(\mathcal{A}) \leq \mathsf{Adv}_{\mathsf{G}}(\mathcal{A}') + \mathsf{negl}(\kappa).$$

We prove the two properties separately. Overall, the proofs follow the same template as the one of the line scheme in Appendix E.2. We, thus, mainly highlight the differences.

Lemma 10. *Assuming the* CE *scheme is correct and modeling* H *as a random oracle, the interval scheme* $\mathsf{CKS}_{\mathsf{Interval}}$ *satisfies integrity.*

Proof. Observe that in the integrity game, $\mathsf{Secret}[\mathrm{u}, \mathsf{e}]$ gets changed in at most three places: First, when it is initialized and, thus, $\perp$ is replaced with either a concrete secret s or a placeholder $\mathfrak{K}_p$. Second, in case the placeholder is replaced by a concrete value and, third, upon erasure. Analogous the respective proof for the line scheme it is, thus, easy to see that $\mathsf{Secret}[\mathrm{u}, \mathsf{e}] \neq \perp$ iff the protocol "knows" key, i.e., iff `*storedAncestor`$(v) \neq \perp$ for $v = \tau.\mathsf{leaves}[\mathsf{e}]$. Further, we observe that the winning conditions in the Append and Grant oracles, as well as the first winning condition in the Accept oracle, are trivially ruled out by corresponding checks in the protocol. It, thus, remains to argue that the second winning condition in Accept as well as the winning condition in Retrieve cannot be triggered.

Next, consider a hybrid experiment $\mathcal{H}_1$ that works like $\mathsf{CKS\text{-}Int}^{\mathcal{A}}_{\mathsf{CKS}_{\mathsf{Interval}}}$ but (a) disables the aforementioned trivial winning conditions, and (b) additionally keeps track of a mapping Ctxt with $\mathrm{Ctxt}[\mathrm{u}, v]$ storing the first (valid) CE ciphertext C_v that user u used for that node, i.e., the first ciphertext u either generated (in Append or Accept) or downloaded and accepted (in Retrieve or Accept). Upon Erase, the value is reset. This modification does not change the game's behavior or winning condition.

Second, consider a hybrid experiment $\mathcal{H}_2$ that does not use placeholders. Instead, in Accept, it defines $\mathsf{Secret}[\mathrm{u}', \mathsf{e}]$ to be the key obtained by decrypting $\mathrm{Ctxt}[\mathrm{u}', \tau.\mathsf{leaves}[\mathsf{e}].\mathsf{parent}]$ (and taking the respective key depending on whether $\tau.\mathsf{leaves}[\mathsf{e}]$ is the left or right child of its parent) instead. Analogous to the line-scheme proof, we observe that this modification can only make the game easier to win $\mathcal{A}$, by triggering an assertion instead of first substituting a placeholder.

Third, consider a hybrid $\mathcal{H}_3$ that upon Accept with an honest grant checks that the accepting user u' and the granting user U agree on the relevant CE ciphertexts of the delegated nodes. That is, the adversary wins if $\mathrm{Ctxt}[\mathrm{u}, v] \neq \mathrm{Ctxt}[\mathrm{u}', v]$ for some node v derivable from a node in the cover V the user sent. Clearly, if $\mathsf{CKS}_{\mathsf{HA}}$.Accept processes the input, then they agree on T_v: Since U' validates the Merkle-Tree on the CE ciphertexts and tags, collision resistance of H thus implies that this check only fails with negligible probability.

Fourth, consider a hybrid $\mathcal{H}_4$ that additionally enforces the invariant that, if set, the key resulting from decrypting $\mathrm{Ctxt}[\mathrm{u}, v.\mathsf{parent}]$ is equal to $\mathsf{Secret}[\mathrm{u}, \mathsf{e}]$ for $v = \tau.\mathsf{leaves}[\mathsf{e}]$, with the adversary winning otherwise. If $\mathrm{Ctxt}[\mathrm{u}, v.\mathsf{parent}]$ is set upon the user generating the ciphertext, then the invariant is maintained due to correctness of the CE scheme. If $\mathrm{Ctxt}[\mathrm{u}', v.\mathsf{parent}]$ is set as part of Accept, there are two options. First, if $\mathsf{Secret}[\mathrm{u}', \mathsf{e}]$ is already set, but $\mathrm{Ctxt}[\mathrm{u}', v.\mathsf{parent}]$ not, then Append must have been called while the sibling's secret has not been known yet. At which point $\mathsf{Secret}[\mathrm{u}', \mathsf{e}] = v.K$ was trivially satisfied. Since as part of Accept, the protocol checks that the new $\mathrm{Ctxt}[\mathrm{u}', v.\mathsf{parent}]$ decrypts to $v.K$ the invariant is hence preserved. Second, if $\mathsf{Secret}[\mathrm{u}', \mathsf{e}]$ gets assigned $\mathsf{Secret}[\mathrm{u}, \mathsf{e}]$, then we know from the invariant introduced in $\mathcal{H}_3$ that the parties have the same ciphertext. Hence, the invariant still holding for the granting party u implies that the invariant is preserved for u' as well.

Finally, we argue that triggering either of the two remaining initial winning conditions in $\mathcal{H}_4$ happens with at most negligible probability. First, consider the check $\mathsf{Secret}[\mathrm{u}', \mathsf{e}] \in \{\perp, \mathrm{s}\}$ failing, where s is the value from $\mathsf{Secret}[\mathrm{u}, \mathsf{e}]$. By our invariants we have however that (1) $\mathrm{Ctxt}[\mathrm{u}, v.\mathsf{parent}] = \mathrm{Ctxt}[\mathrm{u}', v.\mathsf{parent}]$ and (2) for both parties the decryption of their respective ciphertexts equals to the key stored in

Secret. Second, consider the check $\mathsf{Secret}[\mathrm{u}, \mathsf{e}] = \mathrm{secrets}(\mathsf{e})$ in Retrieve being broken. If secrets(e) is set by the protocol as $v.K$, then the check trivially holds, as argued before. If, on the other hand, the protocol derives the key by retrieving the respective ciphertext and decrypt it, note that collision resistance implies that the protocol only accepts the same ciphertext as stored in $\mathrm{Ctxt}[\mathrm{u}, v.\mathsf{parent}]$. Hence, our invariant from $\mathcal{H}_4$ implies the check to not fail either.

Finally, we remark that erasure does not affect integrity, since the integrity notion does not enforce consistency past an erasure and erasing is essentially equivalent to the protocol just "forgetting" all associated state as if it had never learned secrets for the affected periods. □

Finally, we complete by proof by showing the following lemma. The proof, once again, follows closely the one of the line scheme.

Lemma 11. *Assuming the* CE *scheme is correct and modeling* H *as a random oracle, the interval scheme* $\mathsf{CKS}_{\mathsf{Interval}}$ *is secure. That is, for any CKS-compatible game* G*, and any admissible PPT adversary* $\mathcal{A}$*, there exists a PPT adversary* $\mathcal{A}'$ *such that*

$$\mathsf{Adv}_{\mathsf{G}_{\mathsf{CKS}}}(\mathcal{A}) \leq \mathsf{Adv}_{\mathsf{G}}(\mathcal{A}') + \mathsf{negl}(\kappa).$$

Proof. Fix some arbitrary CKS-compatible game G, and an arbitrary PPT adversary $\mathcal{A}$ against $\mathsf{G}_{\mathsf{CKS}}$. In the following we construct an adversary $\mathcal{A}'$ against G that has almost the same advantage as $\mathcal{A}$ against their respective games.

$\mathcal{A}'$ internally runs $\mathcal{A}$ and forwards its interaction with the challenger $\mathcal{C}$. In addition, $\mathcal{A}'$ emulates the various CKS oracles towards $\mathcal{A}$. To this end, $\mathcal{A}'$ keeps track of the various variables such as n and ActualKey[·] of $\mathsf{G}_{\mathsf{CKS}}$, except for $\mathsf{St}[\cdot]$. Instead, it keeps track of the following additional state:

- It maintains a binary tree τ of height h (mainly used for tree math).
- $\mathsf{Secret}[\mathsf{e}]$ stores the e-th secret from the CKS-compatible game G, once known to the reduction.
- $\mathrm{Secret}[\mathrm{u}, \mathsf{e}]$ stores the secret user u used in epochs e. If u uses a key from the CKS-compatible game G that is not known to the reduction yet, it uses $\mathfrak{K}_{\mathsf{e}}$ as a placeholder instead.
- $T[\mathrm{u}, v]$ stores the tag the u used for node v.
- $K[\mathrm{u}, v]$ stores the respective CE key, if known.
- $C[T]$ stores the CE ciphertexts associated with a certain tag.

Using that state, $\mathcal{A}'$ emulates the CKS oracles of $\mathsf{G}_{\mathsf{CKS}}$ as follows:

- **CreateUser:** Keeps track of the new user's existence.
- **Corrupt:** Emulates the $\mathsf{G}_{\mathsf{CKS}}$ oracle by exposing all the epochs the corrupted user knows, learning the actual keys. For each new key s_{e} it learns, it
  - sets $\mathsf{Secret}[\mathsf{e}] \leftarrow \mathrm{s}_{\mathsf{e}}$
  - replaces $\mathfrak{K}_{\mathsf{e}}$ with s_{e} in $\mathrm{Secret}[\mathrm{u}', \mathsf{e}']$ for all u' and e'.

  Consider each u' and each pair of adjacent leaves $v.\mathsf{lChild}$ and $v.\mathsf{rChild}$ (with joint parent v). Let $\mathrm{s}_\ell := \mathrm{Secret}[\mathrm{u}', v.\mathsf{lChild}.\mathsf{leafNumber}]$ and $\mathrm{s}_r := \mathrm{Secret}[\mathrm{u}', v.\mathsf{rChild}.\mathsf{leafNumber}]$, respectively. If both secrets are not placeholders, but for their common parent node v, $K[\mathrm{u}', v]$ has not been set, then we now "open" the fake ciphertext $C[v.T]$. That is, the adversary $\mathcal{A}'$:
  - Sets $K[\mathrm{u}', v] \leftarrow \mathrm{SE.Sim}_2(C[v.T], (\mathrm{s}_\ell \| \mathrm{s}_r))$
  - Copies the key to every other user u'' that uses the same tag, i.e., $T[\mathrm{u}'', v] = T[\mathrm{u}', v]$.
  - Programs the ROM s.t. $\mathsf{RO}[\mathrm{s}_\ell \| \mathrm{s}_r] = K[\mathrm{u}', \mathsf{e}]$

  The reduction $\mathcal{A}'$ then proceeds to open ciphertexts along the path up the tree on the path from v to the root. That is, for each node v' on the path, as long as $K[\mathrm{u}', v'.\mathsf{lChild}]$ and $ceKey[\mathrm{u}', v.\mathsf{rChild}]$ are both set, it sets

  $$K[\mathrm{u}', v'] \leftarrow \mathrm{SE.Sim}_2(C[v'.T], (K[\mathrm{u}', v'.\mathsf{lChild}] \| K[\mathrm{u}', v'.\mathsf{rChild}]))$$

  copies the key to every other u'' using the same tag and programs the ROM.
  Once this opening process is concluded for all users u', $\mathcal{A}'$ assembles a CKS protocol state for U. That is, it copies $K[\mathrm{u}, v]$ and $T[\mathrm{u}, v]$ for all nodes v for which U currently stores state.

– **Append:** $\mathcal{A}$ rejects if the user u already knows a key for the given epoch, i.e., if $\mathsf{Known}[\mathrm{u}, \mathsf{e}] \neq \perp$. If $\mathrm{s} = \perp$, then set $\mathrm{s} \leftarrow \mathsf{Secret}[\mathsf{e}]$ in case that is defined, and $\mathfrak{K}_{\mathsf{e}}$, otherwise. Set $\mathrm{Secret}[\mathrm{u}, \mathsf{e}] \leftarrow \mathrm{s}$. Let $v \coloneqq \tau.\mathsf{leaves}[\mathsf{e}]$ and $v_{\mathsf{sib}} \coloneqq v.\mathsf{sibling}$. If the user doesn't know the secret (placeholder or actual) for the sibling yet, then we are done. Otherwise, $\mathcal{A}'$ proceeds to compact the two nodes as follows. If there exists another user U' that uses the same secrets for those two leaves, then set $T[\mathrm{U}, v.\mathsf{parent}] \coloneqq T[\mathrm{U}, v.\mathsf{parent}]$ and $K[\mathrm{U}, v.\mathsf{parent}] \coloneqq K[\mathrm{U}, v.\mathsf{parent}]$. Observe that this implies $C[T[\mathrm{U}, v.\mathsf{parent}]]$ now being already defined. Else, generate a fresh ciphertext and tag. Assume w.l.o.g. that v is the left child, i.e., that e is odd. (The other case is handled analogously.) If
  - If neither s nor $\mathrm{Secret}[\mathrm{u}, \mathsf{e}+1]$ are placeholders, then set

$$\begin{aligned} K[\mathrm{u}, v.\mathsf{parent}] &\leftarrow \mathsf{RO}[\mathrm{Secret}[\mathrm{u}, \mathsf{e}] \| \mathrm{Secret}[\mathrm{u}, \mathsf{e}+1]] \\ C_{v.\mathsf{parent}} &\leftarrow \mathsf{CE}.\,\mathrm{Enc}(K_{\mathrm{u},\mathsf{e}}, (\mathrm{s} \| K_{\mathrm{u},\mathsf{e}-1})) \\ &\qquad = \mathrm{SE.Enc}(K_{\mathrm{u},\mathsf{e}}, (\mathrm{s} \| K_{\mathrm{u},\mathsf{e}-1})) \end{aligned}$$

    sampling the random oracle if necessary.
  - Otherwise, set $C_{v.\mathsf{parent}} \leftarrow \mathrm{SE.Sim}_1(1^\kappa, 2n)$, where n denotes length of a secret.
  - Compute $T[\mathrm{u}, v.\mathsf{parent}] \leftarrow \mathsf{CE}.\,\mathrm{Tag}(C_{v.\mathsf{parent}}, 0^{2\kappa}) \coloneqq \mathsf{RO}[C \| 0^{2\kappa}]$, sampling the ROM if necessary, and set $C[T[\mathrm{u}, v.\mathsf{parent}]] \leftarrow C_{v.\mathsf{parent}}$.

  Emulate the `*compact` procedure for each node v' along the path from $v.\mathsf{parent}$ to the root as follows:
  - Once the user u is not supposed to know the key for node v', then stop. Otherwise, repeat the following steps.
  - If both $K[\mathrm{u}, v'.\mathsf{lChild}]$ and $K[\mathrm{u}, v'.\mathsf{rChild}]$ are known, then compute the real CE key and tag $K[\mathrm{u}, v]$ and $C_{v'}$.
  - Else, set $C_{v'.\mathsf{parent}} \leftarrow \mathrm{SE.Sim}_1(1^\kappa, 2\ell)$, where this time ℓ denotes length of a CE key.
  - Compute

$$\begin{aligned} T[\mathrm{u}, v'] &\leftarrow \mathsf{CE}.\,\mathrm{Tag}(C_{v'}, T[\mathrm{u}, v'.\mathsf{lChild}] \| T[\mathrm{u}, v'.\mathsf{rChild}]) \\ &\coloneqq \mathsf{RO}[C \| T[\mathrm{u}, v'.\mathsf{lChild}] \| T[\mathrm{u}, v'.\mathsf{rChild}]], \end{aligned}$$

    sampling the ROM if necessary, and set $C[T[\mathrm{u}, v']] \leftarrow C_{v'}$.

  In case of upload = `true`, send an according $\mathrm{st}_{\mathrm{up}}$ to $\mathcal{A}$.
– **Grant:** For granting access to epochs share $= [i, j]$, $\mathcal{A}'$ first emulates the run of the protocol by fetching all the required ciphertexts for deriving $V = \texttt{*cover}(\tau.\mathsf{root}, i, j)$. To emulate the protocol verifying those ciphertexts, $\mathcal{A}'$ accepts them iff the adversary sends back the same ciphertexts U initially generated.
  If leak = `false`, $\mathcal{A}'$ then samples a uniform random handle h and records $\mathrm{Msgs}[\mathrm{h}] \leftarrow (\mathrm{share}, \mathrm{U})$. That is, instead of recording the granting message msg, it simply records user and the share. Then it outputs the handle h.
  If leak = `true`, then $\mathcal{A}'$ invokes $\mathcal{C}.\mathsf{Expose}(\mathsf{e})$ for $\mathsf{e} = i, \ldots, j$ and proceeds analogous to the Corrupt oracle to update K, C, T, and Secret. Finally, it outputs an actual grant message msg.
– **Accept:** If $\mathcal{A}$ delivers a handle h to u', then $\mathcal{A}'$ emulates the Accept oracle using a simple consistency check and copying the relevant information for its maintained state.
  If $\mathcal{A}$ injects a grant message msg, the reduction emulates the performing the consistency checks, if required. (Note that msg contains actual CE keys from which $\mathcal{A}'$ can try to decrypt the ciphertexts served.) If for any of the nodes v' that now gets superseded, U actually did not know the real key yet (but faked a ciphertext) then $\mathcal{A}'$ rejects, as this the probability of $\mathcal{A}$ generating ciphertexts that decrypt to a key unknown to him is negligible. During this process, the reduction also populates $T[\mathrm{U}, \cdot]$, $K[\mathrm{u}, \cdot]$ and $C[T[\mathrm{U}, \cdot]]$ as appropriate. In case no consistency check is performed, the reduction simply stores $T[\mathrm{U}, v]$ and $T[\mathrm{U}, v]$ for each node v contained in the grant message.
– **Retrieve:** The reduction simply simulates the message sent by the corresponding party to the server. Note that this message contains public information only. If by emulating the protocol, $\mathcal{A}'$ learns new ciphertexts, tags, CE keys, or secrets, populate the maintained state accordingly.

– **Erase:** Upon erasure, the reduction emulates any re-derivation of remaining state by either checking that ciphertexts match to one stored in $C[T[\mathrm{u}, v]]$ (if the reduction has seen the corresponding) or actually trying to decrypt and verify tags (in case v was part of an injected granted subtree). If those checks succeed, the reduction purges $\mathrm{Secret}[\mathrm{u}, \mathsf{e}]$ for all involved epochs, and $T[\mathrm{u}, v]$ and $K[\mathrm{u}, v]$ for all involved nodes v.
– **RO:** Upon input $x \in \{0,1\}^*$, $\mathcal{A}'$ returns $\mathsf{RO}[x]$ if that has already been defined. Otherwise, it proceeds as follows:
  - Try to parse x as $\mathrm{s}_{\mathsf{e}-1} \| \mathrm{s}_{\mathsf{e}}$ or $K_{v.\mathsf{lChild}} \| K_{v.\mathsf{rChild}}$, for the respective cases.
    If this succeeds, there exists a user U and a node v such that the query has to be answered consistently to match $\mathsf{CE}.\mathrm{Kg}$ and set RO accordingly. More specifically:
    * If $K[u, v.\mathsf{lChild}] = K_{v.\mathsf{lChild}}$ and $K[u, v.\mathsf{rChild}] = K_{v.\mathsf{rChild}}$, then open C_v using the non-committing property to obtain a consistent K_v and set $\mathsf{RO}[x] = K_v$.
    * If $\mathrm{s}_{\mathsf{e}-1}$ and s_{e} are the secrets used for an even e (as determined by the Test oracle, if necessary), then open C of the node v whose children epochs $\mathsf{e}-1$ and e are. This yields K_v that consistently explains C_v as encryption of $\mathrm{s}_{\mathsf{e}-1}$ and s_{e}. Set $\mathsf{RO}[x] = K_v$.
  - If $\mathsf{RO}[x]$ was not already defined and did not need programming, sample $\mathsf{RO}[x]$ u.a.r. and return that value.

The argument as of why this reduction is successful works analogous to the one for the line scheme. In a nutshell, by the IND-CPA security of the encryption scheme, the fake encryptions are indistinguishable from proper ones, and the non-committing property ensures that we can program the random oracle for $\mathsf{CE}.\mathrm{Kg}$ to retroactively "explain" any opening once the respective message is known. Note in particular that our encryption is not circular but only encrypts up the tree.

The game furthermore ensures that users having consistent information also produce consistent ciphertexts. This is critical as the CE scheme is deterministic while the "faking" algorithm isn't. Using tags as unique identifiers for various state works due to the H (modeled as a ROM) being collision resistant. □