Group Key Progression: Strong Security for Shared Persistent Data

Matilda Backendal [1], David Balbás [2,3], and Miro Haller [4]

[1]ETH Zurich
[2]IMDEA Software Institute
[3]Universidad Politécnica de Madrid
[4]UC San Diego

Abstract

End-to-end encryption allows data to be outsourced and stored on an untrusted server, such as in the cloud, without compromising data privacy. In the setting when this data is shared between a group of users, members also all share access to the same static key material used for data encryption. When the group membership changes, access control is only enforced by the server: security breaches or compelled disclosure would allow even a removed member to decrypt the current shared data.

We propose to move away from static keys and instead use a *group key progression* (GKP) scheme, a novel primitive that enables a dynamic group of users to agree on a persistent sequence of keys while keeping a *compact* local state. GKP ensures that group members can only derive keys within a certain interval of the sequence, a notion that we call *interval access control* (IAC), and also provide post-compromise security. Our GKP construction, called Grappa, combines continuous group key agreement (CGKA, by Alwen et al., 2020) with a new abstraction called *interval scheme*. The latter is a symmetric-key primitive that can derive a sequence of keys from a compact state while preserving IAC. We explore different interval scheme constructions and simulate their storage and communication costs when used in group settings. The most efficient of them is a generalization of dual key regression (Shafagh et al., 2020), which we formalize and prove secure. Overall, our protocols offer a practical and robust solution to protect persistent data shared by a group.

Contents

1 Introduction

The need for stronger cryptographic guarantees for outsourced persistent data (such as encrypted files in cloud storage) has been highlighted by recent attacks, which all identify *sharing* as one of the main challenges [14, 3, 46, 47, 1]. In the related setting of end-to-end encrypted (E2EE) data in transit, a long line of work developed continuous group key agreement (CGKA) [6] to tame the challenge of providing a dynamic group of users with shared key material to use for data encryption. In this paper, we explore how this can be done for persistent data. That is, how can a group of users with access to shared data which is stored encrypted on a server (think a folder of E2EE files) efficiently agree on key material such that they can (1) handle changes in the group constitution, securing future and past data from former and future group members, while (2) retaining access to all of their data, without having to resort to data re-encryption or keeping a linearly growing set of keys. Along the way, we uncover insightful differences between data in transit and data at rest, and show that persistent data necessitates tailored methods to ensure strong end-to-end security.

The state of persistent data. By default, most shared data is now stored in the cloud. This includes encrypted data, such as backups of E2EE messaging groups, files and photos. Modern cryptographic protocols ensure strong security for such data while in transit, for example, by transferring files to the cloud over TLS or using MLS to exchange messages in a group chat. However, few systems in practice uphold these guarantees when the data is stored persistently. In fact, common security models for data in transit, which often aim for strong properties such as forward security (FS) and post-compromise security (PCS), do not consider the confidentiality of data after it reaches the receiver. For example, in cloud backup systems for messaging apps such as WhatsApp [33], the entire message history is E2EE with a static key. This secures the data, but no longer provides any post-compromise security.

A similar issue arises in E2EE cloud storage systems where groups of users share access to a folder of encrypted files. In the threat model of end-to-end encryption, the cloud provider is untrusted, as they may be vulnerable to data breaches or compelled access from authoritarian governments. Hence, one cannot rely on the server to enforce access control. Nonetheless, many current solutions encrypt all files under the same shared key,[1] failing to provide any meaningful cryptographic access control against former group members, nor any PCS guarantees.

Why, then, is there such a gap between the security of E2EE data in transit and at rest?

Forward security and post-compromise security for persistent data. A first observation is that communication applications, including TLS 1.3 [60] and MLS [18], rely on ephemeral keys, which can be discarded once the data has been received. But for persistent data, the data encryption keys must be kept as long as the data is supposed to be accessible—which in practice means until it is explicitly deleted. This invalidates many approaches to provide FS and PCS for data in transit. In messaging, for example, FS is often achieved by symmetric ratcheting, relying on the fact that as soon as a key has been used, it can be safely deleted. PCS is generally achieved by leveraging public-key cryptography to introduce fresh shared randomness in the key derivation. Neither of these approaches are applicable to securing a backup system, or files in a secure shared folder, unless one is willing to store individual keys for every file or message. But this alternative is a non-starter in practice: The storage overhead from the linear growth of the cryptographic state

[1] For instance, the cloud storage provider MEGA (`https://mega.io/`) encrypts the file keys for all files in a shared folder with a static folder key. That folder key is shared with folder members using per-user channels. Access revocation is a non-cryptographic operations that does not rotate this static folder key and instead relies on MEGA to no longer serve former members the ciphertexts of newly uploaded files [56].

of each party in the number of stored messages or files is prohibitive. Alternatives that aim to preserve a sublinear cryptographic state have been studied, such as puncturing [13], addressing FS, and updatable encryption [38, 52, 51, 27, 31]. Unfortunately, these paradigms have important efficiency disadvantages and/or need to rely on weaker security models; we discuss these approaches and their drawbacks more later.

Interval access control. In this work, we aim to present a *deeper understanding of why FS and PCS are fundamentally more challenging to achieve for data at rest.* Our target setting is data storage applications where a group of users share access to outsourced data. Our first contribution is to identify cryptographically-enforced *interval access control* (IAC) as the natural and minimal security notion for this shared persistent data. IAC captures that former group members cannot access data created after their removal, and future members cannot decrypt data created before they were granted access. The first goal of our paper is to *formalize IAC* and *construct a practically efficient protocol* that achieves it.

To motivate the relevance of this notion, we observe that IAC is already present for data in transit.[2] When symmetric ratcheting evolves the key material after every received message to achieve FS, this also means that a user who joins the group later will not be able to read previous messages. In the other direction, messages exchanged after a user left the group are protected because the group key material evolves.[3] Therefore, users are naturally restricted to only have access to messages while they are part of the group.

In contrast to data in transit, achieving IAC for persistent data is notably more challenging because the set of users with access to E2EE data may not only be the current group members. Users joining later may be granted access to some data that was in the collection before they joined, such as the old message history. Moreover, once access is granted, data persistence prevents it from automatically expiring; if the owner of a file or the sender of a message leaves the group, data should remain accessible for the others. In other words, the lifetime of data at rest is decoupled from the membership status of its originator.

Nonetheless, we argue that group membership changes are the natural point to enforce access control. For example, the decision whether or not a messaging user gets access to the conversation history is made when the user joins the group. Similarly, when a shared folder member is removed from the folder, it might be desirable to revoke their ability to decrypt any file that is uploaded after their removal. To achieve this, cryptographic access control for shared E2EE data must be enforced when group changes occur.

A systematic analysis of threat models. Putting our focus on IAC for persistent data comes with a significant shift in threat models compared to the ones for data in transit. IAC protects the group and their data from compromises of individual users, ensuring that the adversary only gains access to a fraction of the data shared by the group (the fraction that the compromised user has access to), and nothing else. Protection against compromises is especially important in E2EE systems, as they shift the security burden to endpoints: compromising an endpoint allows the adversary to bypass all end-to-end guarantees. And studying security in the event of compromises is arguably even more critical for data at rest than data in transit, as the likelihood of compromise increases with the data lifetime.

[2]IAC is sometimes referred to as "secure membership" in messaging literature [16], but it is often not explicitly discussed because it follows from other security guarantees for data in transit. However, this is not the case for data at rest.

[3]Remarkably, this fact holds even for some protocols where post-compromise security is not an explicit goal, such as Sender Keys [16] and Matrix [2]. It is also present in continuous group key agreement (CGKA) [6] used in MLS [18].

Table 1: Systematization of data protection mechanisms for group applications in the case of various data actions (columns) against different adversaries / types of user compromises (rows). We make two assumptions: 1) for data at rest, every user is only granted access to data that was created or updated while they were a group member and 2) no other relevant data event happens between the one mentioned in the column and the compromise. Scenarios marked with blue are the focus and contribution of this paper, cells marked with red have no known protection mechanism, and cells with ¦ŪĒ¦ are only possible in the weaker threat model of updatable encryption.

data event (col) while user has status (row)	**DATA IN TRANSIT**		**DATA AT REST**	
	is sent (but was not yet received)	*is received*	*is created or updated*	*is deleted*
malicious former member	group key rotation (PCS $\Longrightarrow$ IAC)	insecure	group key rotation (IAC)	insecure ¦ŪĒ¦
previously compromised member	pk ratcheting + key rotation (PCS)	insecure	pk ratcheting + key rotation (PCS)	insecure ¦ŪĒ¦
compromised active member	trivially insecure		trivially insecure ¦ŪĒ¦	
not-yet-compromised member	symmetric ratcheting (FS)		insecure ¦ŪĒ¦	puncturing (FS)
malicious future member	symmetric ratcheting[11] (FS $\Longrightarrow$ IAC)		symmetric ratcheting (IAC)	

In Table 1 we present a novel systematic analysis of the mechanisms that protect data at rest and data in transit upon state compromise in a threat model where the adversary can observe all ciphertexts produced by users. Columns contain the actions where transmitted data is sent/received and persistent data is created/deleted. Rows list compromises of honest users happening before, during, and after the data action, as well as fully malicious former/future members. A table cell captures what mechanism (if any) protects the data that undergoes the action specified in the column against an adversary that has the compromise status listed in the row at the time of the action. For instance, the first cell specifies that group key rotation (e.g., in messaging) protects data in transit that is sent by honest users from malicious former members, i.e., users already removed from the group before sending the message. The "is received" column in the same row states that no mechanism can guarantee that data in transit is secure from a member that was part of the group when it was encrypted and sent but removed before the data is received, because we consider a threat model where the adversary can observe all ciphertexts. We provide an extended discussion of Table 1 in Appendix A.

Much of the previous comparison of data at rest and data in transit is reflected in Table 1. Data in transit largely already enjoys IAC thanks to mechanisms like rotating group key material and symmetric ratcheting, as discussed above. (We note, however, that access control is rarely discussed in this setting since it follows "for free" from the mechanisms which are used to achieve FS and PCS.) However, the same mechanisms have not been applied to data at rest before. Instead, research has focused on puncturing to get FS for deleted files and on updatable encryption for PCS, albeit the latter requires a weaker threat model than the one we consider in our table.[12] We mark the scenarios that cannot be protected in our threat model but in which updatable encryption can protect data in a weaker threat model with ¦ŪĒ¦. Most of these scenarios are also unachievable for data in transit,

[11] To reduce complexity, we omit that FS for out-of-order messages is achieved through a key-skipping mechanism similar in flavour to puncturing (see e.g. [19]).

[12] In the threat model of updatable encryption, the adversary only learns the key material of a compromised user, and not the ciphertexts. The server is trusted to not disclose ciphertexts until after the compromise, which allows them to be updated.

marked with red (but to the best of our knowledge there is no threat model relaxation similar to UE for that setting). One notable exception is when persistent data is created or updated by a member that will be compromised later (i.e., in row "not-yet compromised member"). Correctness of a data at rest application mandates that the member still has access to the data at the moment of compromise. Thus, the adversary trivially learns it, unlike for data in transit where the ephemerality of the key material helps to protect the data. Our contributions, highlighted in blue, are to use mechanisms from secure group messaging to achieve IAC and PCS, filling the gap of previously unexplored protections for shared persistent data.

1.1 Our Contributions

We outline the main technical contributions of our work. In what follows, we define time in terms of discrete epochs.

Interval Schemes. In Section 4, we introduce and formalize *interval schemes*, a novel symmetric-key primitive that allows a set of parties to derive a sequence of consistent (epoch) keys from a compact state. A state st enables the derivation of a sequence $(k_a, \ldots, k_b)$ of keys for an interval $[a, b]$. States can be delegated: given a state st for $[a, b]$, it is possible to export a state st' for any $[c, d]$ such that $a \leq c \leq d \leq b$. Correctness ensures that the keys are consistent across all parties. For security, we formalize perfect IAC, which in the context of interval schemes can be expressed intuitively as follows: if the adversary exposes states $st_1, \ldots, st_n$ for intervals $I_1, \ldots, I_n$ respectively, all epoch keys not in $\bigcup_{i \in [n]} I_n$ must be indistinguishable from random. However, we show that for practical efficiency, it is useful to consider a relaxed version of IAC that permits a controlled leakage in between exposed intervals.

Furthermore, we present an efficient instantiation of an interval scheme based on dual key regression (DKR), which was introduced without a formalization in [63]. Our scheme extends the original DKR construction to avoid limitations and achieve security against multiple interval corruptions, which the original scheme did not. Our construction builds on one-way chains OWC, which can be instantiated from a PRG. We use two such chains that run in opposing directions, which we call forward and backward chain. Every chain element is associated to an epoch, and each epoch key can be derived by evaluating the key combiner F on the two corresponding chain elements. Intuitively, given one element on every chain, a user can derive the epoch keys for all epochs between the forward and backward elements as these chains can be advanced easily in one direction. However, epoch keys outside this interval cannot be derived because that would require going in the hard direction on either the forward or backward chain (e.g., for hash chains, that would require breaking the preimage resistance of hash function). Moreover, we show that the key combiner F satisfies a notion we denote by *double-PRF* security, which fills a gap in PRF security notions. We prove that it is implied by dual-PRF security, a result that may be of independent interest for applications that need to combine multiple keys into one. Finally, we also compare the efficiency and security of our DKR-based construction to other interval scheme instantiations, such as using GGM trees.

Why compact state matters. One major advantage of interval schemes (and of GKP, discussed below) is that it allows access control on interval granularity while only having small local states and *no server state*. Compactness is essential for practical efficiency due to the scale of persistent data applications in practice. In some cases, it is possible to wrap a (large) cryptographic state under a single encryption key and outsource the encrypted state to the server, to be downloaded and decrypted every time the user logs in. While this works for single-user cloud storage, it has

several drawbacks for group-based applications. First, the communication complexity for group operations increases significantly, e.g. because a large cryptographic state needs to be sent to newly joining users. Second, always retrieving a cryptographic state from the server increases the potential attack surface for malicious servers.[13] Third, outsourcing state is not possible if the user only has read access to the server, as the user needs to modify its cryptographic state regularly to achieve any advanced security properties. Therefore, decoupling the key material storage from the data storage is a more flexible and practically efficient alternative. Moreover, compact storage may also enable the use of hardware protections—such as a trusted chip that isolates the client state from direct application access—which usually have little storage space.

Group key progression. In Section 5, we present a novel primitive called group key progression (GKP). In a GKP scheme, a group of users can *agree on a sequence of keys across multiple epochs* while maintaining persistent access to them. These keys are consistent across group members' views and can later be used to secure arbitrary data, such as files or backed-up messages. The GKP syntax supports the standard operations for dynamic groups, i.e., adds, removals and key updates. For intuition, GKP is the analog to CGKA for persistent data, where the main novelty lies in GKP's ability to *re-derive past keys from a compact local storage*. Moreover, GKP distinguishes between two member roles: standard members and admins. Admins are users that have full access to all keys and possess additional information that enables them to add or remove members.

For security, we introduce a model based on key indistinguishability games where the adversary has full control over group operations and can corrupt users at any time. The model is parametrized by a predicate and can accurately capture both IAC and PCS for the key sequence for different constructions. We focus on capturing key privacy and do not consider authentication. Hence, we do not allow the adversary to forge or inject messages. As we argue in later sections, authentication can be achieved via signatures and a PKI following previous work for CGKA [7, 17]. We remark that GKP is essentially a key agreement primitive and does not address the confidentiality of data, but only the security for derived keys. These keys can later be used to protect persistent data in different applications (see Section 7).

Grappa: group key progression for persistent access. In Section 6 we provide one practical and efficient instantiation of GKP, that we call Grappa, which combines CGKA and an interval scheme to efficiently generate a sequence of keys to secure persistent data. At a high level, Grappa relies on a small subset of (possibly hundreds of) users—the group admins—to maintain an interval scheme that is unique to every group. These group admins have a private communication channel to synchronize on global interval scheme updates and manage the set of group members. They give group members partial (interval) access to the key sequence. Besides, Grappa relies on CGKA to (1) establish a post-compromise secure communication channel to distribute interval scheme states and extensions, (2) ensure consistent group membership views, and (3) preserve the IAC security of the underlying interval scheme across epochs.

Grappa is constructed with efficiency in mind, keeping the space complexity sublinear in the total number of operations and avoiding expensive per-user operations when distributing new keys. Moreover, the performance of Grappa is independent of the number of persistent data items that are protected with Grappa keys because IAC only needs to be enforced on group membership changes. In summary, Grappa shows that providing IAC for persistent data is feasible with compact

[13] Recent attacks [14, 3, 46] on the cloud storage provider MEGA were based on tampering with such outsourced cryptographic state. They exploit the outsourced state's lack of integrity to modify parts of the encrypted state and observe the client behavior to recover key material. Recent work on key derivation [62] observed that it is surprisingly challenging to authenticate outsourced cryptographic state, as the integrity verification often depends on values stored in the state itself, which leads to a circular dependency.

Table 2: Overview of the assumptions and guarantees of related work on the protection of persistent E2EE data. The column names are in three categories: first, the application targeted by the cited paper(s) [*app*]. Second, the problem setting: [*mal*] trusted (○), partially trusted (◐) or fully malicious (●) server(s); [*one*] relying on a non-colluding subset of servers (○) or only one server (●); [*internal*] relying on a trusted third party (○), a key distribution service or PKI (◐), or implementing everything internally, without external party (●); and [*shared*] considering shared access to data of a dynamic group. Third, security notions: [*FS*] forward security; [*PCS*] post-compromise security; [*IAC*] interval access control; and [*proof*] formal definitions and proofs. We mark that a property is not/partially/fully present with ○/◐/●, respectively.

	info	setting				security			
paper(s)	**app**	**mal**	**one**	**internal**	**shared**	**FS**	**PCS**	**IAC**	**proof**
key regression [39, 15]	FS & efficient key distribution	○	●	○	◐	○	○	◐	●
Droplet [63]	decentralized data access control	●	○	●	●	○	○	○	◐
SiRiUs [41]	E2EE cloud storage	◐	●	◐	●	○	○	○	○
Plutus [48], SUNDR [53], Mylar [59]	secure file sharing (with enc. search)	●	●	◐	●	○	○	○	○
DepSky [24]	distributed E2EE cloud storage	●	○	◐	◐	○	○	○	○
Sieve [65]	access control with ABE	○	●	◐	◐	○	○	●	○
Titanium [32]	metadata-hiding file sharing	●	○	●	●	○	○	●	●
BurnBox [64]	self-revocable encryption	◐	●	◐	○	○	◐	○	●
Puncturable Key Wrapping [13]	FS for symmetric key hierarchies	●	●	○	○	●	○	○	●
CSS [12]	provable E2EE cloud storage	●	●	◐	○	○	○	○	●
Grappa (this paper)	keys for persistent access	◐	●	◐	●	○	●	●	●

states, symmetric primitives for key derivation, and only using public-key cryptography for group management.

2 Related Work

Table 2 gives an overview of related work, including which application they target, what assumptions they make, and the formal guarantees they aim for and prove.

Key regression and updating. Fu, Kamara, and Kohno [39] introduce the term "key regression" to let data publishers enforce cryptographic access control without trusting the content distribution network (CDN). They achieve only one-sided IAC to protect future keys from removed users through the use of a single one-way chain. However, their key rotations do not aim to restrict access to previous keys. Concurrently, Backes, Cachin, and Oprea [15] defined a similar primitive for securely updating keys on access revocation. In both constructions, the owner acts as server and distributes key material to other users. Hence, in Table 2, we mark that these schemes require a fully trusted server. Additionally, [15] proposes lazy re-encryption to protect existing files from removed users. This falls in our definition of IAC for future files, as they are only re-encrypting on file updates. Both [39] and [15] discuss applications involving groups, but the membership management is external to the suggested primitives.

Droplet [63] extended key regression by adding a second one-way chain to construct "dual key regression" for granting interval access. Moreover, they use a binary tree data structure to derive data encryption keys (DEKs). Their system embeds an authorization service in a state machine on a

blockchain which handles group membership operations. Authorized users either receive the binary tree nodes to derive data encryption keys (DEK) for the data that they were granted access to, or they get continuous "subscriber" access. For the latter, Droplet uses dual key regression to wrap DEKs, giving users a forward chain element and then distributing new backward chain elements encrypted with a distribution key via the blockchain. Their system tolerates honest-but-curious servers and a minority of fully malicious blockchain nodes. Dual key regression enables them to get IAC for a single user compromise, limiting the derivable keys to an interval. However, Droplet does not achieve IAC when multiple users are compromised as chains are never rotated: compromising multiple users allows an adversary to derive all intermediate keys between compromised intervals. We address this shortcoming in our generalization of dual key regression. Dual key regression was subsequently used in several further systems [29, 30].

Access control (AC) and encryption for persistent data. A line of work from the system security community (SiRiUs [41], Plutus [48], SUNDR [53], DepSky [24], Mylar [59]) started in the early 2000s to integrate client-side encryption with cloud storage systems to achieve end-to-end security. These pioneering works established the feasibility of E2EE cloud storage but neither aim for security in the presence of compromises nor formally prove their constructions. The scheme with the most advanced security guarantees from this line of work is Sieve [65], where a user encrypts their data with attribute-based encryption and defines an access policy, granting other parties access by distributing keys with the right attributes to them. However, Sieve does not support group access to files with the same permissions as the original owner. Sieve uses key homomorphism to revoke keys by re-encrypting user data on the storage provider. This expensive re-encryption enables Sieve to achieve IAC in a threat model where the server is trusted to honestly perform the re-encryption and delete the old ciphertexts before a compromise, such that the adversary does not learn them.

Titanium [32] hides file sharing metadata with oblivious RAM (ORAM) [42]. They achieve malicious security by performing N-party secure computation with at least one (out of N) trusted server(s). These N servers together maintain an access matrix that explicitly allows users to download files that were shared with them, achieving IAC. For storage, files are secret shared across all servers. A major difference to other works discussed here is that Titanium does not use client-side encryption. If all N servers were to collude, their system would not only fail to provide IAC but the servers could directly decrypt any files without compromising users. Their system comes with a proof in the universal composability framework.

BurnBox [64] implements self-revocable encryption to enable users to temporarily revoke their access to data as protection against compelled access. Restoring access relies on a trusted restoration cache to store a recovery token. The temporary revocation property is an interesting extension for GKP discussed in Section 7. BurnBox partially achieves FS: for anticipated compromises, the user can temporarily revoke their access before the compromise and recover file access afterwards, without the adversary learning the files (as long as the adversary does not learn the recovery token).

Puncturable key wrapping (PKW) [13] achieves forward security for deleted files using puncturable PRFs. Puncturing updates the user's locally stored key such that it can no longer decrypt the ciphertexts for deleted files, achieving forward security even against malicious servers. However, the required key updates of PKW are rather costly. Efficiently achieving FS is an open problem, and out-of-scope for our constructions.

Backendal et al. [12] formalize E2EE cloud storage and introduce the provably secure protocol CSS. While their model and protocol covers file sharing, which is also an application of GKP, they only focus on fundamental confidentiality and integrity guarantees, and not on advanced security properties like IAC or PCS.

Compact Key Storage. Dodis, Jost and Marcedone introduced compact key storage (CKS) [36]

to create backups for secure group messaging. In subsequent work, Dodis and Jost showed how to construct CKS in the standard model [35] with only minor functionality changes. CKS is an addition to a secure messaging application, and focuses on backing up the secrets used to encrypt messages with a compact local storage. Similar to IAC, the more advanced CKS construction allows users to recover messages exchanged in an interval. Despite similar applications, CKS and GKP take very different approaches. CKS is tailored to secure messaging and uses deterministic convergent encryption to derive keys from the encrypted information, which requires that the latter has high entropy. Moreover, CKS requires a server to store all CKS-encrypted secrets in addition to the encrypted messages, requiring linear server storage. GKP, in comparison, derives indistinguishable keys to encrypt arbitrary data without server-side storage of encrypted keys. The security guarantees are also different. CKS conserves FS and PCS guarantees that are already present in the encrypted data (e.g., messaging secrets). We argue that IAC—provided by the indistinguishable keys produced by GKP and which extends to encrypted data—is a more natural security notion for persistent data as FS changes its meaning when messages transition to data at rest: messages accessible to the user at the time of compromise are leaked to the adversary even if they were exchanged before the compromise (cf. Table 1). Hence, the FS achieved by CKS only protects messages in the backup that were explicitly deleted before the compromise.

In summary, CKS is efficient for the specific case of message backups but achieves different security guarantees and GKP is a primitive with more general applications. We provide an extended comparison of CKS to our work in Appendix C, including a description of how GKP can be used for messaging backups and why CKS cannot be used to instantiate DKR or GKP.

Updatable Encryption. The idea of updatable encryption (UE) was first introduced by Boneh et al. [25] and has been extensively studied in the literature [38, 52, 51, 27, 31]. UE allows parties to rotate encryption keys while pushing the re-encryption cost to the server. In short, parties can do that by issuing a token, associated to an old key k and a new key k'. The server can use the token to update a ciphertext that encrypts data under old key k to a new ciphertext encrypted under key k'.

In the UE threat model, the adversary first learns only the key material of a compromised user but not the corresponding ciphertexts. Hence, there is a time window in which ciphertexts can be updated before the adversary gets access to them. This threat model makes the strong trust assumption that the server never discloses old ciphertexts (intentionally or not). This essentially requires secure deletion of old ciphertexts, which is notoriously challenging in practice due to distributed and redundant architecture of storage servers. Our approach instead considers a setting where the full ciphertext history is readily available to the adversary, in which case UE can no longer achieve forward security.

3 Background and Building Blocks

3.1 Notation and Conventions

Functions and variables. We denote assigning value a to variable v with $v \leftarrow a$. Randomly sampling an element e from a space $\mathcal{S}$ is denoted by $e \leftarrow\!\!\$\ \mathcal{S}$. For randomized algorithms F, we write $y \leftarrow\!\!\$\ F(x)$ to denote running F on input x and implicit random coins and assigning the output to y. Similarly, $y \leftarrow F(x)$ for deterministic F. If $F(\cdot)$ returns (a, b, c), then we write $(x, \cdot, z) \leftarrow F(\cdot)$ for $x \leftarrow a$ and $z \leftarrow c$ while ignoring b. We use dot notation to access variables in states. For example, to store the CGKA user state γ_M and the current epoch e in the DKR user state st_U, we write $st_U.(\gamma_M, e) \leftarrow (\gamma_M, e)$. The object field name does not necessarily need to match the name of the

variable being stored, and raw values (such as integers) may also be stored in the state in this way (e.g., $st_U.c \leftarrow 0$). We use pipes (|) to denote different options, e.g., defining a protocol with signature $f_1|f_2(a|b)$ denotes that the following code defines both $f_1(a)$ and $f_2(b)$ (but not $f_1(b)$ or $f_2(a)$, as combinations are defined by the order of options). For sets S, T, we use $S \xleftarrow{\cup} T$ and $S \xleftarrow{-} T$ as a shorthand for $S \leftarrow S \cup T$ respectively $S \leftarrow S \setminus T$. The symbol ε denotes empty objects (e.g., a dictionary or tuple with no elements).

Game-based definitions. We follow the framework of [23]. Games are parameterized by a primitive, and optionally a game bit b, and take an adversary as input. Game oracles are prefixed by $\mathcal{O}$. We write $\mathcal{A}^{\mathcal{O}(\cdot)}(x)$ to denote running the (randomized) adversary $\mathcal{A}$ on input x with access to all game oracles (c.f. Figure 4). Adversaries are implicitly stateful. All subroutines of a game have access to variables defined as **global**. By $\mathrm{G}(\mathcal{A}) \Rightarrow y$ we denote the event that the execution of game G with adversary $\mathcal{A}$ results in output y. We write $\Pr[\mathrm{G}]$ as shorthand for $\Pr[\mathrm{G}(\mathcal{A}) \Rightarrow 1]$. Let $\mathcal{Q}_{\mathcal{O}}(\mathcal{A})$ denote the number of queries to oracle $\mathcal{O}$ by adversary $\mathcal{A}$. The running time of an adversary includes the time for the game procedures to respond to oracle queries. We use the convention that boolean true corresponds to 1 and false to 0. In writing game or adversary pseudocode, it is assumed that boolean variables are initialized to 0, integer variables to 0, and sets tothe empty set $\emptyset$. The symbol $\perp$ stands for "undefined" and is used to implicitly initialize variables and tables, as well as to signal errors. Our algorithms return $\perp$ if they fail or are called on invalid inputs.

List notation. Our pseudo-code uses the data type **list** with Python-inspired syntax. A list l can be initialized with **list**() or by assigning the list elements a_0, a_1, ..., a_r with $l \leftarrow [a_0, a_1, \ldots, a_r]$. Similarly, we unwrap a list and assign variables to list elements with $[a_0, a_1, \ldots, a_r] \leftarrow l$. We denote the number of elements in a list by $|l|$. The notation $l[i{:}j]$ is a shorthand for the sublist $[a_{i \bmod |l|}, a_{(i+1) \bmod |l|}, \ldots, a_{j \bmod |l|}]$, for example $l[2{:}4] = [a_2, a_3, a_4]$ and $l[-1{:}-3] = [a_r, a_1, a_2, \ldots, a_{r-2}]$. The notation $l[i]$ denotes retrieving the element at index $i \bmod |l|$. For example, $l[r] = l[-1] = a_r$. In assignments, $l[i] \leftarrow c$ replaces the element $i \bmod |l|$ with c. Lists can be expanded with $l \xleftarrow{+} [b_1, b_2, \ldots, b_k]$, which results in $l = [a_1, a_2, \ldots, a_r, b_1, b_2, \ldots, b_k]$ for the previous example list l. An exception with an additional shortcut are Integer lists $[a, b]$ for $a, b \in \mathbb{N}$ with $a \leq b$, which denotes the ordered tuple $(a, a+1, \ldots, b)$.

3.2 Building Blocks

Our constructions are built from several standard cryptographic primitives, including PRFs and AEAD. For completeness, we recall their syntax and security in Appendix B. Here, we first introduce one-way chains as a building block for our interval scheme construction, which we can instantiate via either hash chains or via seekable sequential key generators (SSKG) [54, 55]. Next, we describe the original dual key regression scheme that we later generalize to construct an interval scheme. Finally, we recall continuous group key agreement (CGKA) used in Grappa.

3.2.1 One-Way Chains

One-way chains are a useful tool that is found at the core of a vast number of cryptographic protocols. A one-way chain simply consists of a sequence of states $sst_1, sst_2, \ldots, sst_n$ which are related by some one-way function Evolve such that $sst_{i+1} = \mathsf{Evolve}(sst_i)$. To derive any state sst_i, it suffices to know some state sst_j for any $j \leq i$. One possible construction of one-way chains is to use SSKGs that were introduced by Marson and Poettering [54, 55]. We observe that the syntax and security notion of SSKGs also applies (with minor changes) to other constructions such as hash chains. Hence,

borrowing from [55, Defn. 2], we introduce the syntax for one-way chains with the some renaming and the modification that we redefine Seek as the combination of Seek and $\mathsf{SuperSeek}$ ([55, Defn. 5]).

Definition 3.1 (One-way chains)**.** A one-way chain $\mathsf{OWC}_{\ell_s} = (\mathsf{Gen}, \mathsf{Evolve}, \mathsf{Seek}, \mathsf{GetKey})$, parameterized by key length ℓ_s, is a tuple of four algorithms, of which Gen is randomized and the others are deterministic:

- $sst_0 \leftarrow\!\!\$\ \mathsf{Gen}(N)$: given the max chain length N, generates the initial state sst_0.
- $sst_{i+1} \leftarrow \mathsf{Evolve}(sst_i)$: given the state sst_i, outputs the next state sst_{i+1}.
- $sst_{i+j} \leftarrow \mathsf{Seek}(sst_i, j)$: given the state sst_i and $j \in \mathbb{N}$, fast forwards to sst_{i+j}.
- $K_i \leftarrow \mathsf{GetKey}(sst_i)$: given the state sst_i, outputs key $K_i \in \{0,1\}^{\ell_s}$.

For all $k \in \mathbb{N}$, let $\mathsf{Evolve}^k(sst_i)$ denote the k-fold composition of Evolve, i.e., repeatedly applying Evolve to the output of the prior call, starting at sst_i. Then we say that OWC is correct if, for all $N \in \mathbb{N}$, all $sst_0 \leftarrow\!\!\$\ \mathsf{Gen}(N)$, all $i, j \in \mathbb{N}$, and $sst_i \leftarrow \mathsf{Evolve}^i(sst_0)$, we have

$$0 \le i \le i + j < N \implies \mathsf{Evolve}^j(sst_i) = \mathsf{Seek}(sst_i, j) \ .$$

The desirable security guarantee of one-way chains is key indistinguishability in the face of state corruption, i.e., forward security. Again, we can borrow from SSKG and modify the security game of [55, Definition 4] to support multiple users, arriving at the game shown in Figure 1. (We note that the single-user notion—which can be obtained by restricting the adversary to a single query to oracle $\mathcal{O}\mathtt{New}$ in our game—and multi-user notion are equivalent via a standard hybrid argument [20].)

The game has two stages. In the first stage, adversary $\mathcal{A}$ can query oracle $\mathcal{O}\mathtt{New}$ to generate the initial state of a new one-way chain and oracle $\mathcal{O}\mathtt{GetKey}(n, i)$ to expose the i-th key of the n-th chain. The first stage ends when $\mathcal{A}$ outputs a list of exposure indices s_j and challenge indices i_j^*, one for each chain, specifying that it wishes to learn state s_j of the j-th SSKG chain, as well as the i_j^*-key of that chain. The game provides the states s_j for all j and either the real key $K_{j,1}^*$ or a random key $K_{j,0}^*$ based on a game bit b. The adversary wins if it outputs a correct guess of b without having trivially exposed any of the challenge keys. To allow the adversary to corrupt the initial state sst_0^j, it can output -1 in place of a challenge index i_j^*, indicating that it does not want a challenge for chain j.

Definition 3.2 (OWC indistinguishability with forward security)**.** Let OWC be a one-way chain and consider the game $\mathbf{G}_{\mathsf{OWC}}^{\text{ind-fs-}b}$ in Figure 1. The advantage of an adversary $\mathcal{A}$ against the indistinguishability with forward security of OWC is defined as

$$\mathbf{Adv}_{\mathsf{OWC},N}^{\text{ind-fs}}(\mathcal{A}) = \Pr[\mathbf{G}_{\mathsf{OWC},N}^{\text{ind-fs-1}}(\mathcal{A})] - \Pr[\mathbf{G}_{\mathsf{OWC},N}^{\text{ind-fs-0}}(\mathcal{A})] \ .$$

A usual instantiation of one-way chains are *hash chains* with an additional output key. These can be instantiated from a PRG $\mathsf{G}\colon \{0,1\}^{\mathcal{S}} \to \{0,1\}^{\mathcal{S}} \times \{0,1\}^{\ell_s}$. Applying this PRG to a previous state returns the next state and a key as $(sst_{i+1}, k_i) \leftarrow \mathsf{G}(sst_i)$. The key k_i is then returned by $\mathsf{GetKey}(sst_i)$ and leaking it does not compromise the state. It is well known that such hash chains provide forward security upon state corruption, and we refer to e.g. [5, Section 4.2] for a detailed instantiation and a proof. For hash chains, $\mathsf{Seek}(sst_i, j)$ can be implemented by simply calling $\mathsf{Evolve}^j(sst_i)$. However, this operation takes linear time in j. SSKGs improve on this by adding a "fast forward" mechanism which allows to jump from a state to a later one without evolving the

$\mathbf{G}^{\text{ind-fs-}b}_{\mathsf{OWC},N}(\mathcal{A})$:

1 **global** m, $\boldsymbol{X}_k$
2 $((i^*_1, s_1), \ldots, (i^*_m, s_m)) \leftarrow\!\!\$\ \mathcal{A}^{\mathcal{O}(\cdot)}()$
3 **if** $\exists j\colon s_j \leq i^*_j$: **return** 0 // state would expose key
4 **for** $j = 1$ to m **do**:
5 $\quad sst^j_s \leftarrow \mathsf{Seek}(sst^j_0, s_j)$
6 $\quad$ **if** $i^*_j = -1$: $K^*_{j,b} \leftarrow \perp$ // no chall. for instance j
7 $\quad$ **else**:
8 $\quad\quad K^*_{j,1} \leftarrow \mathsf{GetKey}(\mathsf{Seek}(sst^j_0, i^*_j))$
9 $\quad\quad K^*_{j,0} \leftarrow\!\!\$\ \{0,1\}^{\ell_s}$
10 $b' \leftarrow\!\!\$\ \mathcal{A}^{\mathcal{O}(\cdot)}((K^*_{1,b}, sst^1_s), \ldots (K^*_{m,b}, sst^m_s))$
11 **if** $\exists j\colon \boldsymbol{X}_k[j, i^*_j] = 1$: **return** 0 // key exposed
12 **return** b'

$\mathcal{O}\mathtt{New}()$:

13 $m{+}{+}$
14 $sst^m_0 \leftarrow\!\!\$\ \mathsf{Gen}(N)$

$\mathcal{O}\mathtt{GetKey}(n, i)$:

15 $\boldsymbol{X}_k[n, i] \leftarrow 1$
16 $K^n_i \leftarrow \mathsf{GetKey}(\mathsf{Seek}(sst^n_0, i))$
17 **return** K^n_i

Figure 1: Game defining multi-user indistinguishability with forward security of a seekable sequential key generator OWC.

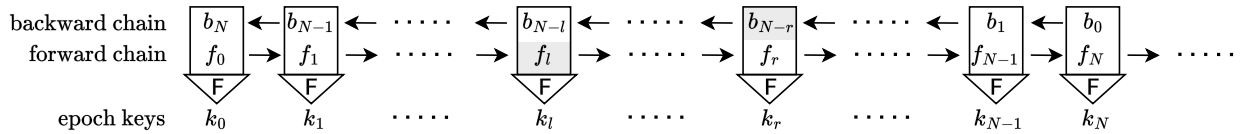

Figure 2: Visualization of DKR with a forward chain with elements f_i for all $i \in [0, \infty]$ and a backward chain with elements b_j for all $j \in [0, N]$. Given f_l and b_{N-r}, the keys for epochs $e \in [l, r]$ can be derived with the key combiner F as $k_e \leftarrow \mathsf{F}(f_e, b_{N-e})$.

state one step at a time. In short, SSKG builds a binary tree from a stateful PRG and states consist of a logarithmic number of internal tree nodes from which successor nodes, and eventually keys associated to tree leaves, can be derived. This efficiency improvement, however, comes at the cost of only supporting bounded-length chains (hence, the parameter N of Gen) and a state size that is logarithmic in the chain length N.

3.2.2 Dual Key Regression

Dual key regression (DKR) was introduced by Shafagh et al. [63] to enforce access control for encrypted data streams. The core idea of DKR, which we illustrate in Figure 2, is simple: a sequence of keys is generated by combining the elements of two one-way chains (e.g., hash chains), running in opposite directions. These are referred to as the *forward* and *backward* chain. In principle, the forward chain can have unbounded length, whereas the backward chain has pre-determined length $N + 1$ for some $N \in \mathbb{N}$. Bounding the backward chain is necessary because the elements are used in the opposite order than they are derived. Hence, once $N + 1$ elements have been used, the chain has arrived at the backward seed b_0, requiring a new chain. To derive the full sequence of keys, it suffices to store the seeds of both chains. Similarly, access to any subinterval $(k_l, k_{l+1}, \ldots, k_r)$ of keys (for $0 \leq l \leq r \leq N$) can be compactly delegated by providing the l-th element f_l from the forward chain and the r-th element b_{N-r} from the backward chain, counting from the back.

3.2.3 Continuous Group Key Agreement

A continuous group key agreement (CGKA) scheme [6] allows a set of parties G, henceforth referred to as a *group*, to agree on a group key that evolves over time and supports changes in

group membership. In CGKA schemes, communication is asynchronous, meaning that not all communicating parties need to be online at the same time. Each party only has a local view of the global protocol state.

CGKA is given by the algorithms described below, which are run locally by the different parties. We follow the syntax from [7] except that we make the slight simplification that welcome messages are not divided into a public and a private part. We remark that each participant has a local state γ which is initialized, and associated to a user identifier *uid*, via $\gamma \leftarrow\!\!\$\ \mathsf{Init}(\mathit{uid})$. We assume that the state γ of the group member with identifier *uid* stores the (possibly empty) set of all group members in $\gamma.G$ and the group key I in $\gamma.I$.

Definition 3.3 (Continuous group key agreement)**.** A CGKA scheme is a tuple of algorithms $(\mathsf{Init}, \mathsf{Create}, \mathsf{Prop}, \mathsf{Commit}, \mathsf{Proc}, \mathsf{Join})$, defined as follows:

- $\gamma \leftarrow\!\!\$\ \mathsf{Init}(\mathit{uid})$: on input a user identifier *uid*, initializes the state of the user, producing the user's initial state γ implicitly storing the *uid* in $\gamma.\mathit{uid}$.

- $\gamma' \leftarrow\!\!\$\ \mathsf{Create}(\gamma)$: on input the caller's state γ, creates a group with only $\gamma.\mathit{uid}$ on it (i.e., setting $\gamma'.G = \{\mathit{uid}\}$) and outputs an updated state γ'.

- $(\gamma', P)/\perp \leftarrow\!\!\$\ \mathsf{Prop}(\gamma, \texttt{ptype}, \mathit{uid})$: on input the caller's state γ, a proposal type `ptype`, and a user identifier *uid* (specifying to which user the proposal refers to), outputs the updated state γ' and a proposal message P, or $\perp$ on failure. Proposal types `ptype` include:

  - `Add` to add user *uid* to the group.
  - `Rem` to remove user *uid* from the group.
  - `Upd` to update the local key material from user *uid*.

- $(\gamma', T, W, I) \leftarrow\!\!\$\ \mathsf{Commit}(\gamma, \overline{P})$: on input the caller's state γ and a (possibly empty) vector of proposal messages $\overline{P}$, outputs the updated state γ', a control message T for existing group members, a welcome message W for new members (without `Add` proposal in $\overline{P}$, we have $W = \varepsilon$), and a new group key I.

- $\gamma'/\perp \leftarrow\!\!\$\ \mathsf{Proc}(\gamma, T)$: on input a the caller's state γ and a control message T, outputs the updated state γ' or $\perp$ if fails.

- $\gamma'/\perp \leftarrow\!\!\$\ \mathsf{Join}(\gamma, W)$: on input the caller's state γ and a welcome message W, the caller joins the group and outputs the updated state γ' or $\perp$ if fails.

We note that all group members, including the creator of a commit message, need to call Proc to apply the changes made by a control message T to their local view of the group. Time in CGKA is usually modeled by *epochs*, which are uniquely associated to a group key. Whenever there is a change in the group, a new group key is established and the group moves to the next epoch.

In most of the literature, as well as in the MLS standard [18], CGKA relies on a delivery service (e.g., a central server) that is usually modeled as either a passive adversary with the ability to corrupt parties arbitrarily [6, 9, 45, 50] or malicious with varying degrees of active capabilities [4, 8, 10, 17]. Additionally, CGKAs usually rely on a trusted authentication service which is implemented via a PKI. In this work, we always refer to the original game-based security model from [6] adapted to the syntax above, which models PCS and FS with respect to a passive adversary.

4 Interval Schemes

We introduce interval schemes, a symmetric key primitive that enables generating a sequence of pseudorandom *epoch keys* from a compact cryptographic state. States can be created on-demand for bounded intervals of epochs, and can later be extended to cover larger intervals. The scheme defines a *global key sequence* that uniquely defines the epoch keys. Thus, different states that cover the same epoch e will derive the same epoch key k_e.

We define the syntax of an interval scheme in Section 4.1, followed by our definition of correctness in Section 4.2 and the security notion that we call interval access control in Section 4.3. Next, in Section 4.4, we present a construction of an interval scheme inspired by dual key regression, which in turn relies on double-PRFs and one-way chains. We describe and prove its security in Section 4.5, Finally, in Section 4.6, we discuss and compare this DKR-based instantation with other interval scheme constructions.

4.1 Interval Scheme Syntax

The main variables that are used in the scheme are: states st, auxiliary information aux, extensions ext, and epoch keys k. The procedure Init initializes the state and produces an auxiliary information value aux. The key sequence at this point is a single key k_0, and so the algorithm sets the epoch e to zero. The state st always contains the information required to derive the epoch keys for a bounded interval of epochs $[l, r]$. Given a state for $[l, r]$, one can export a state for a sub-interval $[l', r'] \subseteq [l, r]$ via the algorithm Export. The auxiliary information aux contains the cryptographic material required to advance epochs (via $\mathsf{Progress}$) and generate future epoch keys.[14] This value can be *rotated* by setting a flag fl in $\mathsf{Progress}$. The set of possible flags is scheme-dependent. Extensions ext contain the information required to extend an interval $[l, r]$ to $[l, r + s]$ covering s additional epochs. These extensions are created via $\mathsf{CreateExt}$ and processed via $\mathsf{ProcExt}$ to merge the extension into a state st. Finally, algorithm $\mathsf{GetKey}(st, e)$ outputs the epoch key of epoch e on input an interval state st covering e.

Definition 4.1 (Interval Scheme)**.** An interval scheme $\mathsf{INT}_{N,\ell,\mathcal{F}} = (\mathsf{Init}, \mathsf{Progress}, \mathsf{Export}, \mathsf{CreateExt}, \mathsf{ProcExt}, \mathsf{GetKey})$ is a tuple of algorithms, parameterized by maximum chain length N, key length ℓ, and set of admissible flags $\mathcal{F}$.

- $(st, aux) \leftarrow\!\!\$\ \mathsf{Init}()$: initializes the state st and an auxiliary information aux, setting $st.e_{max} \leftarrow 0$.
- $(st', aux')/\perp \leftarrow\!\!\$\ \mathsf{Progress}(st, aux, fl)$: on input the state st, the auxiliary information aux, and a progress flag $fl \in \mathcal{F}$ extends the interval covered by st by one epoch (advancing $st.e_{max}$), and returns the updated state st' and the updated auxiliary information aux'.
- $st'/\perp \leftarrow \mathsf{Export}(st, l, r)$: on input the state st and an epoch interval $[l, r]$, returns a new interval state st' covering $[l, r]$ (or $\perp$ if st does not cover $[l, r]$).
- $ext/\perp \leftarrow \mathsf{CreateExt}(st, l, r, s)$: on input the state st, an epoch interval $[l, r]$, and the number of additional epochs s, returns an extension ext to extend states covering the interval $[l, r]$ to $[l, r + s]$ (or $\perp$ if st does not cover $[l, r + s]$).
- $st'/\perp \leftarrow \mathsf{ProcExt}(st, ext)$: on input a state st and extension ext, returns the extended interval state st'.
- $k/\perp \leftarrow \mathsf{GetKey}(st, e)$: on input a state st and an epoch e, returns epoch key k.

[14] This auxiliary information aux can be thought of as a secret value which—in distributed settings or group applications—may only be available to privileged users.

Interval-Agnostic Extensions. Some interval schemes have the useful property that their extensions do not need to depend on the left boundary of the initial interval. This is, if two intervals $[l, r]$ and $[l', r]$ share the same right boundary, the extension of the interval $[l, r]$ can be used to extend the interval $[l', r]$ and vice versa. We capture this formally in Definition 4.2.

Definition 4.2 (Interval-Agnostic Extensions)**.** An interval scheme INT has *interval-agnostic extensions* if for all valid inputs st, l, l', r, and s (i.e., for the epoch intervals $[l, r]$ and $[l', r]$) we have that $\mathsf{CreateExt}(st, l, r, s) = \mathsf{CreateExt}(st, l', r, s)$. In this case, we drop l from the inputs for the syntax of $\mathsf{CreateExt}$.

4.2 Interval Scheme Correctness

Informally, when $\mathsf{Progress}$ is only run on a synchronized state—called the global state—then correctness for an interval scheme guarantees that epoch keys can be derived for all epochs from zero to $st.e_{max}$, and that the keys are consistent w.r.t. the global key sequence, independent of the state used to derive them. That is, once the scheme has progressed to (or past) an epoch e, the associated epoch key k_e must be the same (and not $\perp$) when derived from any state that covers e. Section 4.2 contains the formal correctness definition.

Formally, we define correctness of an interval scheme INT (Definition 4.1) via the game shown in Figure 3. This game captures the correctness requirements by setting the win flag to 1 on violations. This includes when a state covering the last epoch e_{max} cannot be extended (Line 9), keys cannot be derived for covered epochs (Line 16) or are inconsistent (Line 18), states cannot be exported (Line 22) or extensions cannot be created (Line 28), or valid extensions are not merged (Line 35). We say that INT is correct if for all adversaries $\mathcal{A}$, including computationally unbounded ones, $\Pr[\mathbf{G}^{\text{corr}}_{\mathsf{INT}}(\mathcal{A})] = 0$.

4.3 Interval Scheme Security: Interval Access Control

Our next goal is to formalize the IAC security of interval schemes. Intuitively, we want to capture that only parties with access to a state covering epoch e can derive the epoch key k_e. Formally, we ask that epoch keys are indistinguishable from random for an adversary without access to sufficient interval states. In other words, if an adversary has access to $st_1, \ldots, st_n$ for intervals $[l_1, r_1], \ldots, [l_n, r_n]$, all keys not in $\bigcup_{i=1}^{n}[l_i, r_i]$ should be computationally indistinguishable from random. We call this security notion *perfect* IAC. To capture a broader family of schemes, we parametrize the game by a *safety predicate* Safe, which allows one to define the precise security of an interval scheme, which may be weaker than perfect IAC. We formalize this security notion in game $\mathbf{G}^{\text{iac-}b}_{\mathsf{INT},\mathsf{Safe}}$ in Figure 4.

Definition 4.3 (Interval access control of INT)**.** Let INT be an interval scheme and consider game $\mathbf{G}^{\text{iac-}b}_{\mathsf{INT},\mathsf{Safe}}$ in Figure 4. The advantage of an adversary $\mathcal{A}$ against the *interval access control* of INT with respect to predicate Safe is defined as

$$\mathbf{Adv}^{\text{iac}}_{\mathsf{INT},\mathsf{Safe}}(\mathcal{A}) = \left|\Pr[\mathbf{G}^{\text{iac-1}}_{\mathsf{INT},\mathsf{Safe}}(\mathcal{A})] - \Pr[\mathbf{G}^{\text{iac-0}}_{\mathsf{INT},\mathsf{Safe}}(\mathcal{A})]\right| .$$

Game $\mathbf{G}^{\text{iac-}b}_{\mathsf{INT},\mathsf{Safe}}$ has two stages. In both, the adversary has oracle access to the interval scheme algorithms, allowing it to progress the global state with the desired flags, generate states, and export states. The game keeps track of generated intervals and extensions in dictionaries $\boldsymbol{ST}$ resp. $\boldsymbol{E}$. In the first stage, the adversary outputs a challenge epoch e^* and a list I of intervals that it wishes to expose. Additionally, the adversary can obtain the auxiliary value aux corresponding to a given

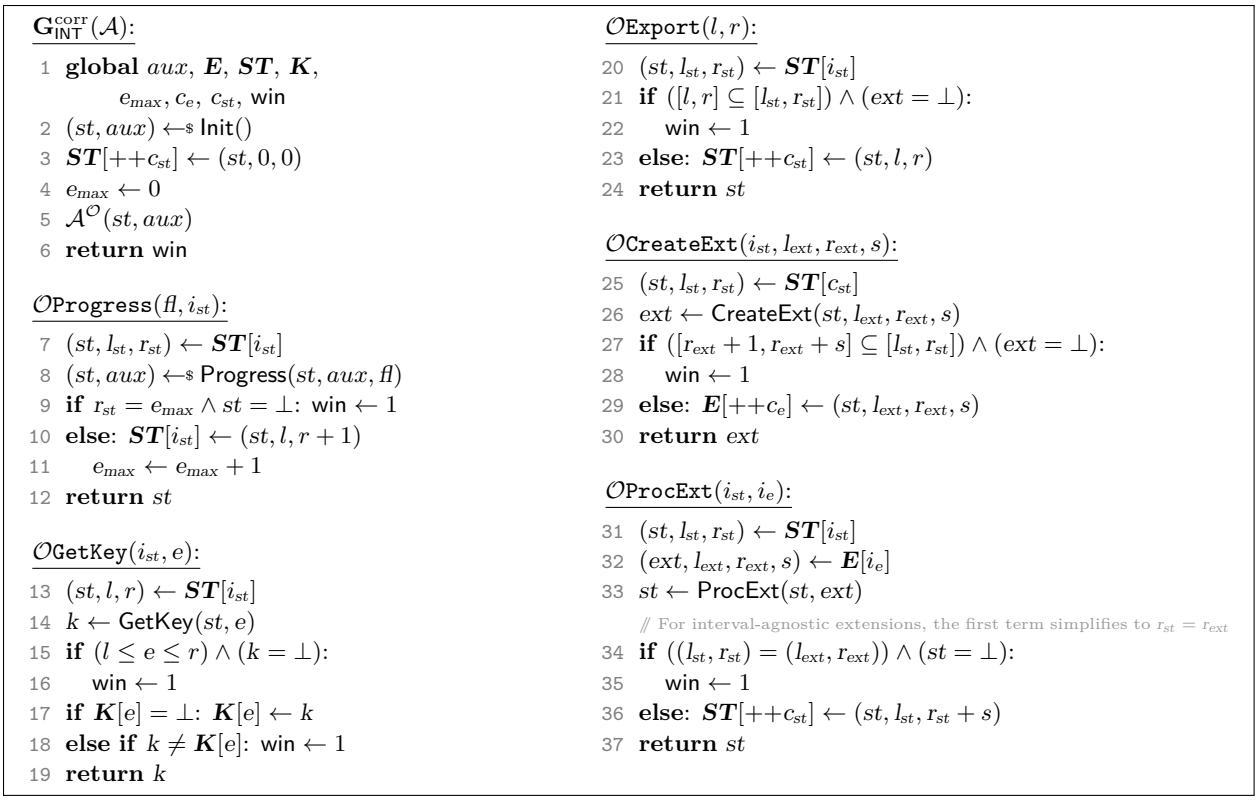

Figure 3: Correctness game for an interval scheme INT.

epoch e by calling oracle $\mathcal{O}$ExposeAux, and expose the key of a given epoch e by calling oracle $\mathcal{O}$GetKey. Dictionaries $\boldsymbol{X}_k$, $\boldsymbol{X}_{st}$ and $\boldsymbol{X}_{aux}$ track the keys, states, and auxiliary values known by the adversary, respectively. In the second stage, the (stateful) adversary gets as input a challenge key k_b corresponding to epoch e^* and a set $\mathcal{S}_{int}$ with the requested states and must output a bit guess b' for whether the k_b was picked at random ($b = 0$) or output by the interval scheme ($b = 1$). To avoid trivial wins, the security of epoch e^* with respect to the compromised intervals and keys is checked on Line 12 of Figure 4 by using two conditions:

- A trivial exposure condition $\neg\boldsymbol{X}_k[e^*] \wedge \neg\boldsymbol{X}_{st}[e^*]$, which returns 0 if the state or key of epoch e^* has been exposed to the adversary through prior queries, tracked in $\boldsymbol{X}_k, \boldsymbol{X}_{st}$.

- The scheme-dependent safety predicate Safe(e^*), which may depend on the flags that are set in the $\mathcal{O}$Progress calls, as well as on the exposures of aux.

The game reflects perfect IAC when Safe$(e^*) =$ **true**. To give an example, this security would be achieved by a "trivial" scheme that samples independent random keys for every epoch, which comes at the cost of state size linear in the number of epochs. We discuss further security/efficiency trade-offs in Section 4.6.

Weakening IAC. The syntax of interval schemes does not specify the set of flags that are admissible in $\mathcal{O}$Progress, and the semantics of the fl parameter are left open. Such flags are meant to enforce on-demand key rotation, and to allow schemes to have smaller states. The DKR-based interval scheme that we introduce in Section 4.4 takes advantage of this.

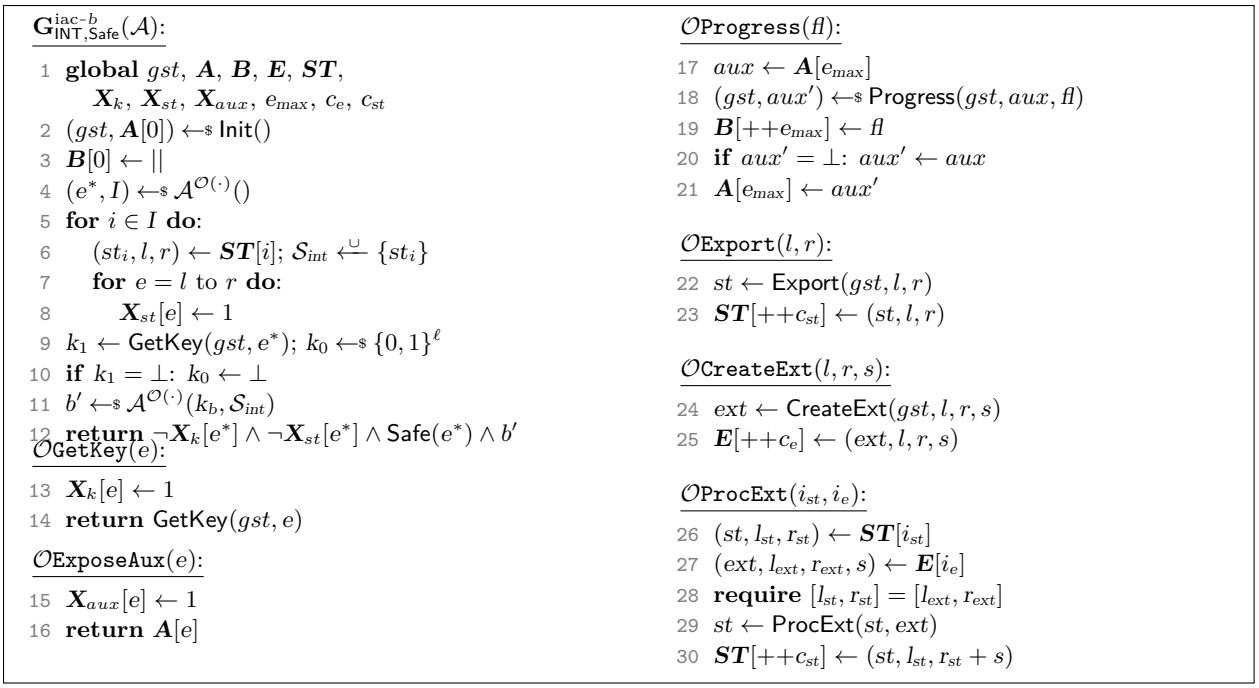

Figure 4: Game defining interval access control of an interval scheme INT.

For further intuition on weak IAC and how to capture it in the safety predicate, consider an example where the adversary has access to states for *disjoint* intervals $[\cdot, l]$ and $[r, \cdot]$. Assume that the scheme design allows for the derivation of the keys between the two intervals unless a key rotation has been enforced by calling Progress on the rotation flag $||$. Then, the safety predicate should be set to $\mathsf{Safe}(e^*) = \textbf{true}$ for $l < e^* < r$ if there has been at least one rotation flag set by Progress in some epoch between the two intervals. Formally,

$$\mathsf{Safe}(e^*) = \forall e_l, e_r : (e_l < e^* < e_r) \wedge (\boldsymbol{X}_{st}[e_l] = \boldsymbol{X}_{st}[e_r] = 1),$$
$$\exists e_{fl} : (e_l < e_{fl} \leq e_r) \wedge (\boldsymbol{B}[e_{fl}] = ||).$$

As an example of what the predicate reflects, if $\mathcal{A}$ has access to state st for $[0, 2]$ and state st' for $[6, 7]$, then keys in the interval $[3, 4, 5]$ are safe only if flag $||$ was set in Progress in at least one epoch $e_{fl} \in [3, 4, 5, 6]$.[15]

4.4 Interval Scheme Construction Based on Dual Key Regression with One-Way Chains and Double-PRFs

In this section, we construct an interval scheme based on DKR, generalizing the original DKR scheme that we described in Section 3.2.2. Our generalization removes the upper limit on the key sequence length by allowing both DKR chains to be replaced, e.g., when they run out of elements. This additionally provides new security properties in the form of hard boundaries for delegatable intervals. This can be used in the group setting to restrict the damage of leaked chain elements. While the security of the original DKR scheme has not been formally analyzed in the literature, we prove the IAC security of our DKR-based interval scheme. This includes the introduction of

[15] In Figure 7 we illustrate the same example for the specific case of our DKR-based interval scheme construction, which is slightly more complex.

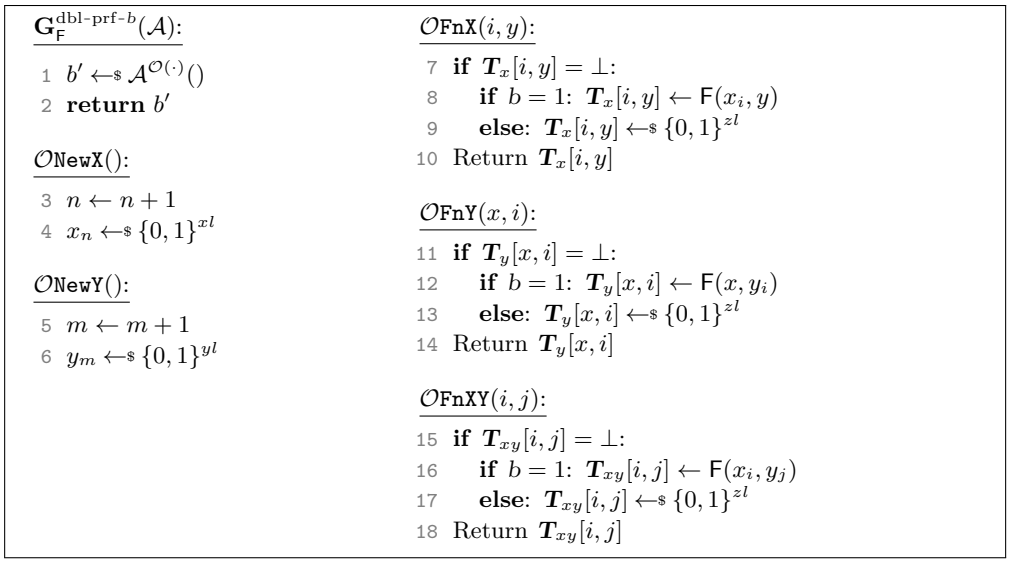

Figure 5: Game defining double-PRF security for $\mathsf{F} : \{0,1\}^{xl} \times \{0,1\}^{yl} \to \{0,1\}^{zl}$.

the *double-PRF* building block in Section 4.4.1, which combines two DKR chain elements, one from the forward chain and one from the backward chain, to derive an epoch key. We prove this construction secure, starting with a formalization of double-PRF security. The one-way chains in our interval scheme can be instantiated either via hash chains or via SSKG. Both constructions have interval-agnostic extensions (see Definition 4.2), but they lead to different cost-storage complexity trade-offs that we analyze in Section 4.6.

4.4.1 Double-PRFs.

A function family $\mathsf{F} : \{0,1\}^{xl} \times \{0,1\}^{yl} \to \{0,1\}^{zl}$ is a function with two arguments, usually called key and message. For each function family F, the *swapped* function $\bar{\mathsf{F}} : \{0,1\}^{yl} \times \{0,1\}^{xl} \to \{0,1\}^{zl}$ is defined as $\bar{\mathsf{F}}(y,x) = \mathsf{F}(x,y)$. That is, $\bar{\mathsf{F}}$ swaps the key for the message, and vice versa. We say that F is a secure pseudorandom function (PRF) if, for any uniformly random $x \in \{0,1\}^{xl}$, $\mathsf{F}(x,\cdot)$ is indistinguishable from a random function $f : \{0,1\}^{yl} \to \{0,1\}^{zl}$ for a computational adversary that does not know x. Following [22], we call F a *swap-PRF* if $\bar{\mathsf{F}}$ is a secure PRF. The formal definitions are in Appendix B.1.

If F is both a PRF and a swap-PRF, it is often called a dual-PRF [22]. In the following, we need a slightly stronger security assumption, which demands that F is secure as both a PRF and a swap-PRF *simultaneously*. That is, it must be indistinguishable from a random function when keyed either by the first input or the second input, as well as when keyed through both inputs at the same time. We call this new notion *double-PRF* security.

Formally, double-PRF security is defined through the game $\mathbf{G}_{\mathsf{F}}^{\text{dbl-prf}}$ in Figure 5, where the adversary has oracle access to both F (through oracle $\mathcal{O}\texttt{FnX}$) and $\bar{\mathsf{F}}$ (through oracle $\mathcal{O}\texttt{FnY}$). Security is in the multi-user setting [20] and the adversary can generate keys via oracles $\mathcal{O}\texttt{NewX}$ (for new elements in $\{0,1\}^{xl}$ which act as keys for F) and $\mathcal{O}\texttt{NewY}$ (for elements in $\{0,1\}^{yl}$ keying $\bar{\mathsf{F}}$). To maintain consistency in the ideal world, the game stores values returned by oracles $\mathcal{O}\texttt{FnX}$ and $\mathcal{O}\texttt{FnY}$ in tables $\boldsymbol{T}_x$ and $\boldsymbol{T}_y$, respectively. The separation into two tables ensures that in the ideal world, $\mathsf{F}(x_i,\cdot)$ and $\bar{\mathsf{F}}(y_j,\cdot)$ are modeled as independent random functions, even if the adversary happens to make queries $\mathcal{O}\texttt{FnX}(i,y)$ and $\mathcal{O}\texttt{FnY}(x,j)$ such that $x_i = x$ and $y_j = y$. Finally, the adversary has access to an oracle $\mathcal{O}\texttt{FnXY}$ which takes two key indices i,j as input. When the game bit $b=1$, the

oracle returns $\mathsf{F}(x_i, y_j)$ for x_i and y_j generated by oracle $\mathcal{O}\texttt{NewX}$ and $\mathcal{O}\texttt{NewY}$, respectively. In the ideal world (when $b = 0$), the oracle returns independent, uniformly random points in $\{0,1\}^{zl}$, with consistency maintained through table $\boldsymbol{T}_{xy}$.

Definition 4.4 (Double-PRF security)**.** Consider game $\mathbf{G}_{\mathsf{F}}^{\text{dbl-prf}}$ in Figure 5. The double-PRF advantage of an adversary $\mathcal{A}$ is given by

$$\mathbf{Adv}_{\mathsf{F}}^{\text{dbl-prf}}(\mathcal{A}) = \Pr[\mathbf{G}_{\mathsf{F}}^{\text{dbl-prf-1}}(\mathcal{A})] - \Pr[\mathbf{G}_{\mathsf{F}}^{\text{dbl-prf-0}}(\mathcal{A})] \ .$$

The following lemma states that any dual-PRF is also a double-PRF. Concretely, this means that we can instantiate a double-PRF with e.g. HMAC [21], which was recently shown to be a dual-PRF [11]. The proof is in Appendix D.

Lemma 4.5 (prf $\wedge$ swap-prf $\iff$ dbl-prf)**.** A function family F: $\{0,1\}^{xl} \times \{0,1\}^{yl} \to \{0,1\}^{zl}$ is a double-PRF if and only if it is a PRF and a swap-PRF.

4.4.2 Construction $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$

We are now ready to give our interval scheme construction. Let N' and ℓ be the maximum chain length and output key length parameters, respectively. Let OWC be a one-way chain with key length ℓ_s and let F: $\{0,1\}^{\ell_s} \times \{0,1\}^{\ell_s} \to \{0,1\}^{\ell}$ be a double-PRF. We construct a DKR-based interval scheme $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$, from OWC and F as shown in Figure 6 and explained below.

$\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ **data types.** Our $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ instantiation uses *fs* and *bs* to refer to the seed of a forward or backward chain, respectively. As the $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ construction maintains multiple such chains, we refer to their seeds by indexing (e.g., $fs_0, fs_1, \ldots$). The $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ states keep two lists, one for forward chains and one for backward chains that both store epoch-chain pairs, sorted by epochs in ascending order. For forward chains, (e_{fs}, fs) denotes the chain with seed *fs*, starting at epoch e_{fs}. This chain can derive keys for *later* epochs $e \geq e_{fs}$ until the next forward chain or the maximum chain length $e_{fs} + N'$. For backward chains, (e_{bs}, bs) denotes the chain with seed *bs* at epoch e_{bs}. This chain can derive keys for *earlier* epochs $e \leq e_{bs}$ until the previous backward chain or epoch 0. Note that elements on the backwards chain are released in reverse order of derivation, and the maximal chain length defines the number of future backward chain elements until a new chain must be created (at the latest, in epoch $e_{bs} + 1$). We refer to the backward and forward chains that are used to derive keys for the maximum epoch e_{max} as the *current chains*.

$\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ **subroutines.** The algorithms in Figure 6 use two helper functions $\mathsf{GetFChains}$ and $\mathsf{GetBChains}$. They take as input a list of forward resp. backward chains, as well as a left and right epoch interval boundary l and r. They return the list of seeds necessary to derive all forward resp. backward chain elements in the interval $[l, r]$. Helper $\mathsf{GetFChains}$ advances the forward chain seed to the state of epoch l to prevent disclosing chain elements needed to derive keys for epochs $e < l$ (Line 53 in Figure 6). Similarly, $\mathsf{GetBChains}$ advances the backward chain seed to the state of epoch r to prevent disclosing chain elements needed to derive keys for epochs $e > r$ (Line 57 in Figure 6). Note that both functions assume well-formed inputs (i.e., $0 \leq l \leq r \leq e_{max}$), which is guaranteed by the calling functions.

Using the above helper functions, the algorithms of $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ work as follows.

$\mathsf{Init}()$**:** Initializes the DKR state *st*. By calling $\mathsf{Progress}$ with a flag $||$, the current maximum epoch is set to $e_{max} = 0$ and the lists for forward and backward seeds $\mathcal{F}, \mathcal{B}$ are populated with initial chains and the auxiliary value *aux* is set to the backward chain seed.

$\mathsf{Progress}(st, aux, \mathit{fl} \in \{\varepsilon, \mapsto, \leftarrow\!\!\shortmid, ||\})$**:** This algorithm increases the interval covered by state st by one, advancing e_{max}. Recall that epoch key k_e for epoch e is only part of the global key sequence after $\mathsf{Progress}$ advanced e_{max} past epoch e. Before that, a call to $\mathsf{Progress}$ can move to different chains before we use, for example, all elements on the backwards chain. Releasing a new element of the backward chain requires access to the backward chain seed bs for some epoch $e_{bs} > e_{max}$ as these elements are released in reverse order. This seed bs is provided in the auxiliary information aux unless the backward chain ran out of elements. If new chains are required, either because one of the current chains reached its maximal length N' or because it was explicitly requested with a flag fl, algorithm $\mathsf{Progress}$ calls $\mathsf{OWC.Gen}$ to initialize a new one-way chain of length N'. The supported flags for $\mathsf{Progress}$ are $\mapsto$ for a new forward chain, $\leftarrow\!\!\shortmid$ for a new backward chain, and $||$ for both. If $\mathit{fl} \in \{\leftarrow\!\!\shortmid, ||\}$, then the algorithm will also return the new backward chain seed in aux.

$\mathsf{Export}(st, l, r)$**:** Uses the helper functions to retrieve the (possibly multiple) forward and backward chains seeds for deriving all keys in the interval $[l, r]$, and sets the interval boundaries $st.l$ and $st.r$ of the exported state accordingly.[16] The helper functions $\mathsf{GetFChains}$ and $\mathsf{GetBChains}$ use the seek operation $\mathsf{OWC.Seek}$ to advance the first forward chain to epoch l and the last backward chain to epoch r, which ensures that the chains in the exported state cannot derive any keys outside of $[l, r]$.

$\mathsf{CreateExt}(st, r, s)$**:** Creates an interval-agnostic extensions extension for epochs $[r + 1, r + s]$ that can extend interval $[l, r]$ to $[l, r + s]$ for any left boundary l. Extensions are almost identical to interval states in our construction, except that the extension often does not include the first forward chain seed. This makes extensions more compact and is possible because existing chains of the extended interval already contain the necessary forward chain seed to derive elements of the extension's first forward chain unless epoch $r + 1$ is the start of a new forward chain.

$\mathsf{ProcExt}(st, ext)$**:** To extend a state st with extension ext, algorithm $\mathsf{ProcExt}$ merges the lists of forward and backward chains. If the extension contains a later element of the same backward chain that is used to derive the last epoch key of the extended interval, then only the later element is stored in the returned state st for compactness (Lines 37–40 of Figure 6).

$\mathsf{GetKey}(st, e)$**:** To retrieve the key k_e for an epoch e, algorithm GetKey runs the helper functions $\mathsf{GetFChains}$ and $\mathsf{GetBChains}$ to retrieve the relevant chains for epoch e from the state st. From these chains, algorithm GetKey extracts the forward key fk and backward key bk using $\mathsf{OWC.GetKey}$. These keys are then combined with the double-PRF F (cf. Section 4.4.1) to compute the returned epoch key $k_e = \mathsf{F}(fk, bk)$.

[16] Note that all of our algorithms operate on a local "copy" of their arguments. When Export modifies st, this does not change the calling user's interval state but rather returns a new state that is a copy of the state passed in the argument of Export with the modifications done in that function.

$\underline{\mathsf{Init}(\varepsilon)}$:

1 $st.(e_{max}, \mathcal{F}, \mathcal{B}) \leftarrow (-1, \mathbf{list}(), \mathbf{list}())$
2 $(st, aux) \leftarrow\!\!\$\ \mathsf{Progress}(st, \varepsilon, ||)$ // set up chains
3 **return** (st, aux)

$\underline{\mathsf{Progress}(st, aux, \mathit{fl})}$:

// retrieve current forward/backward chain epoch
4 **if** $|st.\mathcal{F}| > 0$: $(e_{fs}, \cdot) \leftarrow st.\mathcal{F}[-1]$
5 **else**: $e_{fs} \leftarrow 0$
6 **if** $aux \neq \varepsilon$: $(e_{bs}, bs) \leftarrow aux$
7 **else**: $e_{bs} \leftarrow 0$
8 $aux \leftarrow \varepsilon$ // suppress returning unchanged aux
9 $st.e_{max} \leftarrow st.e_{max} + 1$
// check if new chains are needed
10 **if** $\mathit{fl} \in \{\mapsto, ||\} \vee st.e_{max} \geq e_{fs} + N'$:
11 // move to new forward chain
12 $st.\mathcal{F} \xleftarrow{+} [(st.e_{max}, \mathsf{OWC.Gen}(N'))]$
13 **if** $\mathit{fl} \in \{\mapsto, ||\} \vee st.e_{max} > e_{bs}$:
14 // move to new backward chain
15 $bs \leftarrow \mathsf{OWC.Gen}(N')$
16 $e_{bs} \leftarrow st.e_{max} + N' - 1$
17 $aux \leftarrow (bs, e_{bs})$
18 $st.\mathcal{B} \xleftarrow{+} [(\varepsilon, \varepsilon)]$ // to be filled in on Line 21
19 // extend current backward chain to +1 element
20 $bs \leftarrow \mathsf{OWC.Seek}(bs, e_{bs} - st.e_{max})$
21 $st.\mathcal{B}[-1] \leftarrow (st.e_{max}, bs)$
22 **return** (st, aux)

$\underline{\mathsf{Export}(st, l, r)}$:

23 **pre**: $0 \leq l \leq r \leq st.e_{max}$
24 $st.\mathcal{F} \leftarrow \mathsf{GetFChains}(st.\mathcal{F}, l, r)$
25 $st.\mathcal{B} \leftarrow \mathsf{GetBChains}(st.\mathcal{B}, l, r)$
26 $st.(l, r) \leftarrow (l, r)$
27 **return** st

$\underline{\mathsf{CreateExt}(st, r, s)}$:

28 **pre**: $0 \leq r + 1 \leq r + s \leq st.e_{max}$
29 $ext \leftarrow \mathsf{Export}(st, r + 1, r + s)$
30 // omit first forward chain unless extension starts with a new forward chain seed.
31 **if** $\nexists (r + 1, \cdot) \in st.\mathcal{F}$:
32 $ext.\mathcal{F} \leftarrow ext.\mathcal{F}[1{:}-1]$
33 **return** ext

$\underline{\mathsf{ProcExt}(st, ext)}$:

34 **pre**: $st.r + 1 = ext.l$
35 $st.\mathcal{F} \xleftarrow{+} ext.\mathcal{F}$
36 // discard last backward chain if on new bs' chain
37 $(e, bs) \leftarrow st.\mathcal{B}[-1]$
38 $(e', bs') \leftarrow ext.\mathcal{B}[0]$
39 **if** $\mathsf{OWC.Seek}(bs', e' - e) = bs$:
40 $st.\mathcal{B} \leftarrow int.\mathcal{B}[0{:}-2]$
41 $st.\mathcal{B} \xleftarrow{+} ext.\mathcal{B}$
42 $st.r \leftarrow ext.r$
43 **return** st

$\underline{\mathsf{GetKey}(st, e)}$:

44 **pre**: $e \in [st.l, st.r]$
// Get forward chain elem to derive key for epoch e
45 $[(\cdot, \mathit{fs})] \leftarrow \mathsf{GetFChains}(st.\mathcal{F}, e, e)$
46 $\mathit{fk} \leftarrow \mathsf{OWC.GetKey}(\mathit{fs})$
// Get backward chain elem to derive key for epoch e
47 $[(\cdot, bs)] \leftarrow \mathsf{GetBChains}(st.\mathcal{B}, e, e)$
48 $bk \leftarrow \mathsf{OWC.GetKey}(bs)$
49 $k_e \leftarrow \mathsf{F}(\mathit{fk}, bk)$ // key combiner, cf. Section 4.4.1
50 **return** k_e

$\underline{\mathsf{GetFChains}(\mathcal{F}, l, r)}$:

// get the forward seeds necessary to derive keys for epochs in $[l, r]$.
51 $[(e_0, \mathit{fs}_0), (e_1, \mathit{fs}_1), \ldots, (e_{|\mathcal{F}|-1}, \mathit{fs}_{|\mathcal{F}|-1})] \leftarrow \mathcal{F}$
52 **let** i, $j \in [0, |\mathcal{F}| - 1]$ s.t. $e_i \leq l < e_{i+1}$ and $e_j \leq r < e_{j+1}$ for $e_{|\mathcal{F}|} = e_{|\mathcal{F}|-1} + N'$
// Seeking avoids disclosing the forward chain state needed to derive keys for epochs $l' < l$.
53 $\mathcal{F}[i] \leftarrow (l, \mathsf{OWC.Seek}(\mathit{fs}_i, l - e_i))$
54 **return** $\mathcal{F}[i{:}j]$

$\underline{\mathsf{GetBChains}(\mathcal{B}, l, r)}$:

55 $[(e_0, bs_0), (e_1, bs_1), \ldots, (e_{|\mathcal{B}|-1}, bs_{|\mathcal{B}|-1})] \leftarrow \mathcal{B}$
56 **let** i, $j \in [0, |\mathcal{B}| - 1]$ s.t. $e_{i-1} < l \leq e_i$ and $e_{j-1} < r \leq e_j$ for $e_{-1} = -1$
// Seeking avoids disclosing the backward chain state needed to derive keys for epochs $e > r$.
57 $\mathcal{B}[j] \leftarrow (r, \mathsf{OWC.Seek}(bs_j, e_j - r))$
58 **return** $\mathcal{B}[i{:}j]$

Figure 6: Instantiation of a dual key regression scheme $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]_{N',\ell}$ from a one-way chain OWC with a maximum chain length N' and ℓ-bit keys, and a double-PRF F.

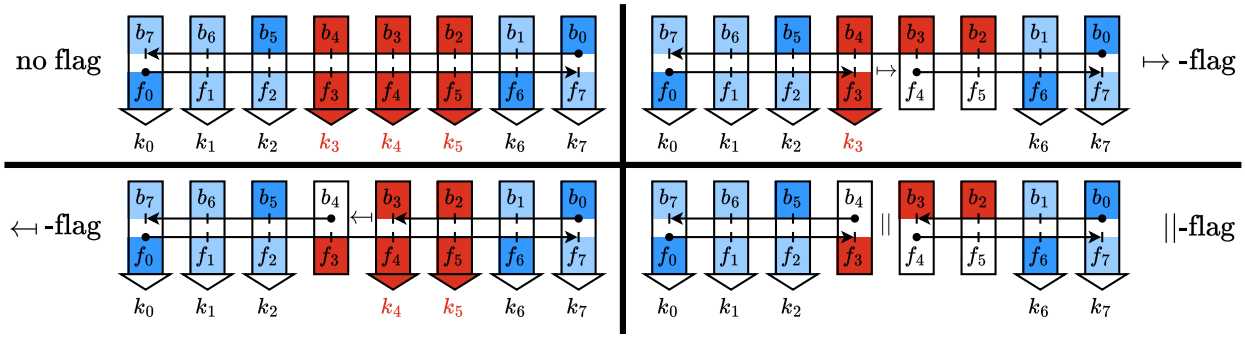

Figure 7: Example of how a progress flag set when the interval is extended to epoch 4 affects the keys derivable from states for intervals $[0, 2]$ and $[6, 7]$ containing (f_0, b_5) and (f_6, b_0), respectively. It depicts the impact of the following flags, in left-to-right, top-to-bottom order: none, new forward chain ($\mapsto$), new backward chain ($\leftarrow$), both new chains ($||$). The chain elements necessary to derive the keys for intervals $[0, 2]$ and $[6, 7]$ are marked blue (the interval state elements are darker, derived values a lighter shade of blue) and additionally leaked elements are marked red. All keys that can be derived have a triangle and are listed below the chains.

4.5 Security of $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$.

Intuitively, the IAC security of our DKR-based interval scheme follows from the one-wayness of the chains for a single interval: the state containing f_l and b_{N-r} gives access to the keys for epochs $e \in [l, r]$, as per Figure 2, but not, for instance, to the key k_{l-1} since element f_{l-1} is not derivable from f_l. However, this guarantee no longer holds against adversaries with access to *multiple* states.

As can be seen in Figure 7, access to two (or more) non-overlapping states (f_0, b_{N-2}) and (f_6, b_{N-7}) for $N = 7$ gives access not only to keys (k_0, k_1, k_2) and (k_6, k_7), but also to keys (k_3, k_4, k_5). The reason is that each state leaks partial information about epochs outside of the interval; as long as a chain continues uninterrupted, a single element suffices to derive all later elements in the chain.

To control this leakage and to avoid giving access beyond intended epochs, our construction supports chain rotation. In our example, switching to a new chain at epoch 4 has the following effect (illustrated in Figure 7):

1. *New forward chain* (progress flag $\mathit{fl} = \mapsto$): forward chain elements from epoch 4 and later are not accessible with a state ending before epoch 4. In the example, epoch keys k_4 and k_5 remain secure.

2. *New backward chain* ($\mathit{fl} = \leftarrow$): backward chain elements before epoch 4 are not accessible with a state starting at epoch 4 or later. In the example, epoch key k_3 remains secure. Note that this also generates a new *aux*.

3. *New backward and forward chain* ($\mathit{fl} = ||$): all k_3, k_4, k_5 remain secure.

Additionally, note that the compromise of *aux* leaks the entire backward chain, potentially affecting future keys until a new backward chain is created. We state the corresponding IAC safety predicate in Figure 8 and formalize security as follows. The full proof is deferred to Appendix E.

The following theorem states that $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ is a secure DKR scheme if OWC provides indistinguishability with forward security and F is a double-PRF.

Theorem 4.6 (IAC security of $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$)**.** Let OWC be a one-way chain and let F be a function family and let $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$, parametrized by maximum chain length N' and output key length ℓ, be defined in pseudocode of Figure 6. Let $\mathcal{A}$ be an adversary against the IAC security of $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$

$$\mathsf{Safe}(e^*) = \mathsf{SafeFC}(e^*) \vee (\mathsf{SafeBC}(e^*) \wedge \mathsf{SafeAux}(e^*)) .$$
$$\mathsf{SafeFC}(e^*) = \forall e_x < e^* : \boldsymbol{X}_{st}[e_x],\ \exists e_{fl} : (e_x < e_{fl} \leq e^*) \ \wedge\ (\boldsymbol{B}[e_{fl}] \in \{\mapsto, ||\}).$$
$$\mathsf{SafeBC}(e^*) = \forall e_x > e^* : \boldsymbol{X}_{st}[e_x],\ \exists e_{fl} : (e^* < e_{fl} \leq e_x) \ \wedge\ (\boldsymbol{B}[e_{fl}] \in \{\mapsfrom, ||\}).$$
$$\mathsf{SafeAux}(e^*) = \forall e_x : \boldsymbol{X}_{aux}[e_x], \exists e_{fl} : (\boldsymbol{B}[e_{fl}] \in \{\mapsfrom, ||\}) \ \wedge\ \big((e^* < e_{fl} \leq e_x) \vee (e_x < e_{fl} \leq e^*)\big).$$

Figure 8: Additional safety predicate Safe for the security of our $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ scheme. The predicate SafeFC captures forward chain safety at the challenge epoch e^*: there must be a forward chain replacement in between any exposed epoch and e^*. Similarly, SafeBC captures backward chain safety: there must be a backward chain replacement in between e^* and the next exposed epoch. Finally, $\mathsf{SafeAux}$ captures backward chain safety upon full leakage, i.e., upon leakage of the auxiliary value aux, which in $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ is the backward chain seed. As the entire backward chain leaks if aux is exposed, any exposure must occur either (a) before a backward chain replacement that occurs before e^*, or (b) after a backward chain replacement that occurs after e^*.

with respect to Safe in Figure 8. Then we can construct adversaries $\mathcal{B}_{\mathsf{OWC}}$, $\mathcal{B}_{\mathsf{F}}$ such that

$$\mathbf{Adv}^{\mathrm{iac},\mathsf{Safe}}_{\mathsf{D}[\mathsf{OWC},\mathsf{F}]}(\mathcal{A}) \leq \mathbf{Adv}^{\text{ind-fs}}_{\mathsf{OWC},N}(\mathcal{B}_{\mathsf{OWC}}) + \mathbf{Adv}^{\text{dbl-prf}}_{\mathsf{F}}(\mathcal{B}_{\mathsf{F}}) .$$

Adversary $\mathcal{B}_{\mathsf{OWC}}$ makes at most $2 \cdot \mathcal{Q}_{\mathcal{O}\texttt{GetKey}}(\mathcal{A})$ queries to oracle $\mathcal{O}\texttt{GetKey}$ and at most $2 + 2 \cdot \mathcal{Q}_{\mathcal{O}\texttt{Progress}}(\mathcal{A})$ queries to oracle $\mathcal{O}\texttt{New}$. Adversary $\mathcal{B}_{\mathsf{F}}$ makes at most 1 query to each of its oracles. The running times of both are roughly that of $\mathcal{A}$.

On a high level, the proof consists of two main steps. The first is a reduction to the ind-fs security of the one-way chain OWC, in which the forward and backward chain keys at the challenge epoch are replaced by random. The reduction simulates the progression of the global state, and when adversary $\mathcal{A}$ outputs its challenge epoch and interval exposure requests, the reduction requests the states needed to simulate the exposed intervals. In particular, for all chains except the two which cover the challenge epoch, the reduction exposes the seeds. Due to the safety predicate $\mathsf{Safe}(e^*)$, either the forward or backward chain that covers the challenge epoch e^* is guaranteed to not be exposed at epoch e^*. This allows the reduction to request a challenge at epoch e^* in the OWF game, replacing the corresponding forward (fk) and/or backward chain key (bk) by random. The second step is a reduction to the double-PRF security of the PRF F, which replaces $\mathsf{F}(fk, bk)$ by random. We defer the full proof to Appendix E.

4.6 Interval Scheme Comparison and Efficiency

We compare our DKR-based construction of an interval scheme to other approaches and identify three distinguishing properties of our construction: compact storage, low communication complexity, and interval-agnostic extensions. These properties are crucial for scalability in the group setting, as we discuss in Section 6.

Baseline: trivial interval scheme. The simple scheme $\mathsf{trivial}$ achieves IAC by sampling a fresh key for every epoch. States simply contain the set of consecutive keys in an interval (hence, have linear storage size), and aux is empty.

SSKG-based DKR. We can instantiate OWC in $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ from an SSKG scheme S as described in Section 3.2.1. We denote this construction by $\mathsf{D}[\mathsf{S}, \mathsf{F}]$.

Hash chain-based DKR. Similarly, if we instantiate OWC from a hash chain H as described in Section 3.2.1, we obtain a construction $\mathsf{D}[\mathsf{H}, \mathsf{F}]$.

Interval Schemes from GGM Trees. A different approach to build an interval scheme is to use a Goldreich-Goldwasser-Micali (GGM) tree [43, 44] of length N, which recursively applies a

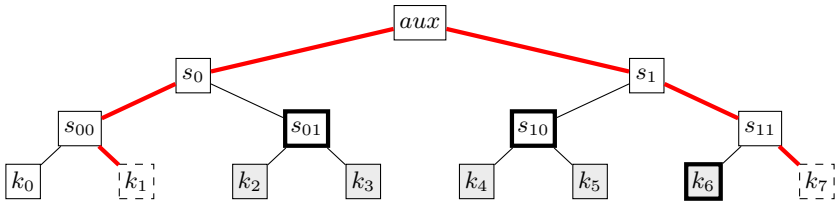

Figure 9: States on a GGM tree. The keys k_e for epoch $e \in [2, 6]$, highlighted in light gray, can be derived from the state $st = (s_{01}, s_{10}, k_6)$. The state contains as the right/left co-path of the first node to the left/right of the interval, respectively, up until the point where the two paths intersect (in this example, at the root). The starting nodes of the path are marked with a dashed box and their path to the root is **bold and red**, the selected states on the co-path have bold boxes.

---

length-doubling PRG $G : \{0,1\}^\ell \to \{0,1\}^{2\ell}$ for inputs of length ℓ to build a binary tree where the child nodes are set to the first and last ℓ bits of the output from applying G to the parent node state. The tree leaves then correspond to epoch keys. To construct an interval scheme, we can set the auxiliary value aux as the root of the tree. Figure 9 shows an example of a global key sequence of $N = 8$ keys derived from aux. For example, the ℓ-bit states s_{00} and s_{01} are derived from s_0 by applying the PRG: $s_{00}||s_{01} \leftarrow G(s_0)$. Access to subintervals of the key sequence can be granted through a set of internal tree nodes that allow for the derivation of all leaves between the endpoints of the interval, and no leaves outside that. For instance, in Figure 9, the state st for keys k_e for $e \in [2, 6]$ contains the minimal set of tree nodes (s_{01}, s_{10}, k_6) that allows us to derive all keys. Interval extensions must include the node or set of nodes that allow one to derive the next epoch key.

We identify two possible constructions that we denote by GGM1 and GGM2, depending on how they handle extensions. GGM1 simply provides the leaf corresponding to the next epoch. Such extensions are interval-agnostic but result in an interval state size that is linear in the number of covered epochs. GGM2 avoids this state growth by leveraging the tree structure to create extensions while preserving IAC, which makes extensions dependent on the left endpoint of the interval. For example, for the GGM tree in Figure 9, consider two states $st = (s_{01}, s_{10}, k_6)$ and $st' = (s_0, s_{10}, k_6)$ (i.e., which contain the tree nodes required to derive keys k_e for $e \in [2, 6]$ respectively $e \in [0, 6]$). Now, if we want to extend st' to k_7 while preserving its compactness, we would simply send the root aux. However, st cannot be sent aux as this also reveals k_0 and k_1, breaking IAC. Hence, the extension for st must contain only s_1.

Interestingly, both GGM-based constructions achieve perfect IAC inside a single tree (i.e., there is no leakage beyond the interval bounds). However, to handle leakage of aux and support an unbounded number of keys, this construction still needs to support rotating to new trees with a different aux root, similar to chain rotations of DKR. Such rotations can be captured via a progress flag $fl = ||$. The safety predicate for both constructions is the following:

$$\begin{aligned}\mathsf{Safe}(e^*) = \exists e_l, e_r : (\boldsymbol{B}[e_l] = \boldsymbol{B}[e_r] = ||) \ \wedge \ (e_l \leq e^* < e_r) \\ \wedge \ (\forall e \in [e_l, e_r], \boldsymbol{X}_{aux}[e] = 0).\end{aligned}$$

Moreover, GGM1 allows for puncturing to achieve forward security for deleted epoch keys—as first noted by [26, 28, 49]—although at the cost of degrading the state size to be linear in the number of punctured keys. To the best of our knowledge, it is not known how to achieve FS with logarithmic storage size, and hence this is also not supported in GGM2.

Practical efficiency. The optimal choice of a scheme may vary on the application, as concrete costs depend on usage patterns. Table 3 summarizes the comparison including asymptotic metrics of

Table 3: Comparison of the trivial, GGM, SSKG, and hash-based instantiations of interval schemes for ℓ-bit keys, global key sequence length N, and $u \ll N$ distinct intervals. We report the worst-case cost asymptotically but omit the $\mathcal{O}(\cdot)$ notation. The columns denote the size of the auxiliary value ($|aux|$), of a state ($|st|$), of the extensions to expand all u intervals by one epoch ($|ext|$); the computation cost of forwarding to future states (Seek); whether **IAC** requires no explicit blocks (○), blocks for both aux and st compromise (◐), or only on aux compromise (◔); whether it supports secure deletion with (◐) or without (●) state size degradation; and whether it supports interval-agnostic extensions (●). We highlight the best values for each column.

	$\|aux\|$	$\|st\|$	$\|ext\|$	Seek	**IAC**	**secure del.**	**int. agnostic**
trivial	0	$\ell \cdot N$	ℓ	1	●	●	●
GGM1	ℓ	$\ell \cdot N$	ℓ	$\log(N)$	◔	◐	●
GGM2	ℓ	$\ell \cdot \log(N)$	$u \cdot \ell$	$\log(N)$	◔	○	○
D[S, F]	ℓ	$\ell \cdot \log(N)$	ℓ	$\log(N)$	◐	○	●
D[H, F]	ℓ	ℓ	ℓ	N	◐	○	●

the costs of each scheme for a *single* pair of chains respectively a single GGM tree. The hash-based DKR D[H, F] is best suited for our group application Grappa due to its compact storage (independent of the chain length N) and interval-agnostic extensions. We will see in Section 6.4 that this is crucial to keep communication complexity low—which enables efficient and scalable protocols—as it avoids costly pairwise channels in the group setting, and preserves the advantages of the well-established CGKA design such as the delivery service.

Furthermore, we argue that although D[H, F] spends time linear in N to derive chain elements, this is not a concern in practice as evaluating hash functions is a fast operation on modern hardware and N is of moderate size.

5 Group Key Progression

Interval schemes enable the derivation of a sequence of keys that can later be shared and restricted to intervals. While they may find other applications, our main motivation for introducing interval schemes is the group setting. However, interval schemes are a symmetric key primitive that lacks the support for the group operations that are expected for any group key agreement primitive. In this section, we introduce *group key progression* (GKP), which is a standalone primitive that enables a (dynamic) group of users to privately share a sequence of keys across time. GKP can be seen as the analogue of CGKA [6] for the persistent setting. Importantly, GKP does not rely on external (e.g., server-side) storage, and only requires a server for message delivery (as CGKA does). To establish a syntax and security model for GKP, we naturally take inspiration from CGKA.

5.1 Group Key Progression Syntax

In a GKP scheme, time is measured in epochs e, which are the time between two group actions, i.e., changes to the group membership or key rotations. Every epoch has an associated key called the *epoch key*. The GKP syntax supports the standard operations for dynamic groups, i.e., adds, removals and key updates. GKP schemes distinguish between standard members and group admins. Only admins are allowed to add or remove other users, and their cryptographic state may contain additional key material compared to that of a standard member similar to prior work on group messaging [17, 57].

Definition 5.1 (Group key progression)**.** A GKP scheme GKP = (InitUser, Create, ExecCtrl, ProcCtrl, JoinCtrl, GetEpochKey) is a tuple of algorithms that we define below. Associated to the scheme is a space of users $\mathcal{U} = \{uid_1, \ldots, uid_u\}$.

- $st_U \leftarrow\!\!\$\ \mathsf{InitUser}(uid)$; initializes the state of a new user with identifier $uid \in \mathcal{U}$.
- $st'_U \leftarrow\!\!\$\ \mathsf{Create}(st_U)$; on input the user state st_U, creates a group owned by the calling user and outputs an updated user state st'_U.
- $(st'_U, \tau, \omega) \leftarrow\!\!\$\ \mathsf{ExecCtrl}(st_U, cmd, \text{arg})$; on input the calling user's state st_U, optional argument set arg (e.g., the target user's identifier uid), this algorithm executes a control command cmd from the list below. This outputs a control message τ, a welcome message ω, and updates the user state st'_U. The welcome message is only used for command `Add` and will be set to $\perp$ otherwise.

  The following commands cmd are supported *for group admins*:

  - `Add`: add user $uid \in$ arg to the group with access from the current epoch.
  - `Rem`: remove non-admin user $uid \in$ arg from the group.
  - `AddAdm`: grant existing user $uid \in$ arg admin privileges for the group.
  - `RemAdm`: revoke the admin privileges of user $uid \in$ arg.
  - `UpdAdm`: refresh the state st_U of the calling admin.
  - `RotKeys`: rotates the key material of the entire group.

  The following command cmd is supported *for non-admin members*:

  - `UpdUser`: performs an update to refresh the local state of the caller.
- $st'_U/\perp \leftarrow \mathsf{ProcCtrl}(st_U, \tau)$; on input a user state st_U and a control message τ, the user processes the control message, evolves to the next epoch (if needed), and returns its updated state st'_U, or $\perp$ if execution fails.
- $st'_U/\perp \leftarrow \mathsf{JoinCtrl}(st_U, \omega)$; on input a user state st_U and a welcome message ω, the user processes the welcome message to join the group. It returns the updated state st'_U or $\perp$ if execution fails.
- $K_e/\perp \leftarrow \mathsf{GetEpochKey}(st_U, e)$; given the user state st_U and an epoch e, derive the corresponding epoch key K_e, or $\perp$ if the user cannot derive this key.

Implicitly, the users in a GKP scheme rely on a service provider to send messages to other parties. As usual in the messaging literature (e.g. [6, 50, 45, 16]), the service provider is assumed to provide correct message delivery. A temporary unavailability of the service provider may degrade correctness but not security. Similar to messaging protocols, GKP does not have any server-side global state. Hence, we can rely on the delivery service to handle concurrent operations.

Correctness. In a correct GKP scheme, all parties processing a consistent sequence of control messages (i.e. share consistent views of the state) must obtain the same epoch keys from $\mathsf{GetEpochKey}$ for all epochs available to them. We omit the formal definition as it is similar to interval scheme correctness (see Section 4.2).

5.2 Group Key Progression Security Model

Consistent with the security models established for group key agreement primitives [6, 58], the GKP security model considers an adversarial service provider (often a central server in practice) that can

schedule group operations and messages sent by parties, as well as corrupt users at any point in time. Hence, the service provider can collude with corrupted parties to gain access to secret data.

Our model primarily captures key indistinguishability of the GKP epoch keys. We remark that GKP security should not be interpreted as a file sharing security model, since 1) GKP is a key agreement primitive, agnostic to the use that is later given to the keys, and 2) the GKP service provider only handles message delivery and can be decoupled from a server storing ciphertexts. For applications such as cloud storage and message backups, other security notions such as data integrity and user authentication must be considered. This is out of scope of this work, as these properties should be provided with additional security mechanisms robust against malicious servers, as done for example in [12].

Security game. The security of a group key progression scheme GKP is defined via a key indistinguishability game (see Definition 5.2) running between a challenger and an adversary $\mathcal{A}$. After an interaction with multiple oracles, $\mathcal{A}$ wins the game if it guesses a real-or-random bit b correctly and if the so-called *cleanness predicate* C allows the sequence of oracle calls made by $\mathcal{A}$.

Definition 5.2 (Key indistinguishability of GKP)**.** Let GKP be a group key progression scheme. We define *key indistinguishability* with respect to a cleanness predicate C for GKP via the game $\mathbf{G}^{\text{kind-}b}_{\mathsf{GKP},\mathsf{C}}$ in Figure 10. We define the advantage of an adversary $\mathcal{A}$ in $\mathbf{G}^{\text{kind-}b}_{\mathsf{GKP},\mathsf{C}}$ as

$$\mathbf{Adv}^{\text{kind}}_{\mathsf{GKP},\mathsf{C}}(\mathcal{A}) = \Pr[\mathbf{G}^{\text{kind-1}}_{\mathsf{GKP},\mathsf{C}}(\mathcal{A})] - \Pr[\mathbf{G}^{\text{kind-0}}_{\mathsf{GKP},\mathsf{C}}(\mathcal{A})] \,.$$

We say that GKP is (q, t, ε)-kind$_{\mathsf{C}}$ secure if for all adversaries $\mathcal{A}$ running in time at most t and making at most q oracle queries we have $\mathbf{Adv}^{\text{kind}}_{\mathsf{GKP},\mathsf{C}}(\mathcal{A}) \leq \varepsilon$.

Time and synchronization between parties in the security game is modeled by integer *epochs*, which match the GKP epochs except for standard member updates, as these do not generate a new epoch key. The game enforces that each generated control message is associated to a unique epoch. We say that a user *uid* is at epoch $e > 0$ if it belongs to the group at epoch e, and the last control message processed by *uid* corresponds to e. Additionally, we introduce a global epoch game variable ep, which stores the epoch of the latest control message that was issued (not necessarily processed). For each new control message, ep is incremented by 1. The number of member updates is kept in a counter up.

Table 4 describes several dictionaries tracked by the challenger during the game. Note that, except for the state dictionary $\boldsymbol{ST}$ and the key dictionary $\boldsymbol{K}$, the content of all dictionaries is known to the adversary.

Oracles. We informally describe the oracles that the adversary has access to in the game in Figure 10.

$\mathcal{O}\mathtt{Init}(uid)$**:** Initializes a user *uid* via $\boldsymbol{ST}[uid] \leftarrow \mathsf{InitUser}(uid)$.

$\mathcal{O}\mathtt{Deliver}(uid, e)$**:** *uid* processes the control message $\boldsymbol{T}[e]$. If the processing succeeds, sets the epoch of user *uid* to $\boldsymbol{E}[uid] \leftarrow e$, except if *uid* is removed from the group; then it sets $\boldsymbol{E}[uid] \leftarrow \perp$.

$\mathcal{O}\mathtt{Join}(uid, e)$**:** *uid* processes the join message $\boldsymbol{W}[e]$. If the processing succeeds, sets $\boldsymbol{E}[uid] \leftarrow e$.

$\mathcal{O}\mathtt{Expose}(uid)$**:** Leaks the state $\boldsymbol{ST}[uid]$ of the user *uid* to $\mathcal{A}$.

$\mathcal{O}\mathtt{GetKey}(e)$**:** Leaks the epoch key $\boldsymbol{K}[e]$ for epoch e to $\mathcal{A}$.

$\mathcal{O}\mathtt{Test}(e)$**:** The adversary gets a real-or-random key corresponding to epoch e, depending on whether the game bit b is 0 or 1. This oracle can only be called once.

Game $\mathbf{G}^{\text{kind-}b}_{\mathsf{GKP},\mathsf{C}}(\mathcal{A})$:

global $\boldsymbol{G}, \boldsymbol{AD}, \boldsymbol{E}, \boldsymbol{T}, \boldsymbol{W}, \boldsymbol{K}, \mathsf{ep}, \mathsf{up}$
$(uid_0 \in \mathcal{U}) \leftarrow \mathcal{A}^{\mathcal{O}\mathtt{Init}(\cdot)}()$
$\boldsymbol{ST}[uid_0] \leftarrow \mathsf{Create}(\boldsymbol{ST}[uid_0])$
$\boldsymbol{G}[0] \leftarrow \{uid_0\}$; $\boldsymbol{E}[uid_0] \leftarrow 0$
$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathcal{O}(\cdot)}()$
if C **return** b' **else return** $\bot$

$\mathcal{O}\mathtt{Init}(uid \in \mathcal{U})$:

$\boldsymbol{ST}[uid] \leftarrow \mathsf{InitUser}(uid)$

$\mathcal{O}\mathtt{Deliver}(uid \in \mathcal{U}, e)$:

$st \leftarrow \mathsf{ProcCtrl}(\boldsymbol{ST}[uid], \boldsymbol{T}[e])$
if $st = \bot$: **return**
if $\boldsymbol{K}[e] = \bot$:
  $\boldsymbol{K}[e] \leftarrow \mathsf{GetEpochKey}(st, e - \mathsf{up})$
$\boldsymbol{ST}[uid] \leftarrow st$
if $uid \in \boldsymbol{G}[e]$: $\boldsymbol{E}[uid] \leftarrow e$
else $\boldsymbol{E}[uid] \leftarrow \bot$

$\mathcal{O}\mathtt{Join}(uid \in \mathcal{U}, e)$:

$st \leftarrow \mathsf{JoinCtrl}(\boldsymbol{ST}[uid], \boldsymbol{W}[e])$
if $st = \bot$: **return**
$\boldsymbol{ST}[uid] \leftarrow st$; $\boldsymbol{E}[uid] \leftarrow e$

$\mathcal{O}\mathtt{Expose}(uid \in \mathcal{U}, e)$:

if $e = \boldsymbol{E}[uid]$ **return** $\boldsymbol{ST}[uid]$

$\mathcal{O}\mathtt{GetKey}(e)$:

return $\boldsymbol{K}[e]$

$\mathcal{O}\mathtt{Test}(e \leq \mathsf{ep})$:

$K_0 \leftarrow\!\!\$\ \{0,1\}^{\ell}$; $K_1 \leftarrow \boldsymbol{K}[e]$
return K_b

$\mathcal{O}\mathtt{Add}(uid \in \mathcal{U}, uid' \in \mathcal{U})$:

pre: $\mathsf{ep} = \boldsymbol{E}[uid] \ \wedge\ \boldsymbol{T}[\mathsf{ep}] = \bot$
$(\boldsymbol{T}[\mathsf{ep}], \boldsymbol{W}[\mathsf{ep}], \boldsymbol{ST}[uid])$
  $\leftarrow \mathsf{ExecCtrl}(\mathtt{Add}, \boldsymbol{ST}[uid], uid')$
if $\boldsymbol{T}[\mathsf{ep}] \neq \bot$:
  ep++; $\boldsymbol{G}[\mathsf{ep}] \xleftarrow{\cup} \{uid'\}$

$\mathcal{O}\mathtt{Remove}(uid \in \mathcal{U}, uid' \in \mathcal{U})$:

pre: $\mathsf{ep} = \boldsymbol{E}[uid] \ \wedge\ \boldsymbol{T}[\mathsf{ep}] = \bot$
$(\boldsymbol{T}[\mathsf{ep}], \cdot, \boldsymbol{ST}[uid])$
  $\leftarrow \mathsf{ExecCtrl}(\mathtt{Rem}, \boldsymbol{ST}[uid], uid')$
if $\boldsymbol{T}[\mathsf{ep}] \neq \bot$:
  ep++; $\boldsymbol{G}[\mathsf{ep}] \xleftarrow{-} \{uid'\}$

$\mathcal{O}\mathtt{AddAdm}|\mathcal{O}\mathtt{RemoveAdm}(uid \in \mathcal{U}, uid' \in \mathcal{U})$:

pre: $\mathsf{ep} = \boldsymbol{E}[uid] \ \wedge\ \boldsymbol{T}[\mathsf{ep}] = \bot$
$(\boldsymbol{T}[\mathsf{ep}], \cdot, \boldsymbol{ST}[uid]) \leftarrow$
$\mathsf{ExecCtrl}(\mathtt{AddAdm}|\mathtt{RemAdm}, \boldsymbol{ST}[uid], uid')$
if $\boldsymbol{T}[\mathsf{ep}] \neq \bot$:
  ep++
  $\boldsymbol{AD}[\mathsf{ep}] \xleftarrow{\cup} \{uid'\} | \boldsymbol{AD}[\mathsf{ep}] \xleftarrow{-} \{uid'\}$

$\mathcal{O}\mathtt{Update}(uid \in \mathcal{U})$:

pre: $\mathsf{ep} = \boldsymbol{E}[uid] \ \wedge\ \boldsymbol{T}[\mathsf{ep}] = \bot$
$(\boldsymbol{T}[\mathsf{ep}], \cdot, \boldsymbol{ST}[uid]) \leftarrow$
$\mathsf{ExecCtrl}(\mathtt{UpdUser}, \boldsymbol{ST}[uid], \bot)$
if $\boldsymbol{T}[\mathsf{ep}] \neq \bot$: ep++; up++

$\mathcal{O}\mathtt{UpdateAdm}|\mathcal{O}\mathtt{RotKeys}(uid \in \mathcal{U})$:

pre: $\mathsf{ep} = \boldsymbol{E}[uid] \ \wedge\ \boldsymbol{T}[\mathsf{ep}] = \bot$
$(\boldsymbol{T}[\mathsf{ep}], \cdot, \boldsymbol{ST}[uid]) \leftarrow$
$\mathsf{ExecCtrl}(\mathtt{UpdAdm}|\mathtt{RotKeys}, \boldsymbol{ST}[uid], \bot)$
if $\boldsymbol{T}[\mathsf{ep}] \neq \bot$: ep++

Figure 10: Security game $\mathbf{G}^{\text{kind-}b}_{\mathsf{GKP},\mathsf{C}}$ between a challenger and an adversary $\mathcal{A}$, parametrized by cleanness predicate C.

Table 4: Summary of dictionaries tracked by the $\mathbf{G}^{\text{kind-}b}_{\mathsf{GKP},\mathsf{C}}$ challenger.

$\boldsymbol{G}[\cdot]$	maps epoch e to the set of group members at epoch e
$\boldsymbol{AD}[\cdot]$	maps epoch e to the set of group admins at epoch e
$\boldsymbol{ST}[\cdot]$	maps user *uid* to their current state (updated on oracle calls on *uid*)
$\boldsymbol{T}[\cdot]$	maps epoch e to the unique control message of e
$\boldsymbol{W}[\cdot]$	maps epoch e to the unique welcome message of e (if applicable)
$\boldsymbol{E}[\cdot]$	maps user *uid* to their local epoch (by convention, $\bot$ for non-members)
$\boldsymbol{K}[\cdot]$	maps epoch e to its corresponding epoch key

$\mathcal{O}\texttt{Add}(uid, uid')$, $\mathcal{O}\texttt{Remove}(uid, uid')$ **:** User uid adds/removes uid'. If successful, the oracle stores the control message in $\boldsymbol{T}[\mathsf{ep}]$ (and the welcome message in $\boldsymbol{W}[\mathsf{ep}]$ for $\mathcal{O}\texttt{Add}$), advances the global epoch $\mathsf{ep}{+}{+}$ and updates the set of group members for the new epoch $\boldsymbol{G}[\mathsf{ep}]$ by adding/removing uid'.

$\mathcal{O}\texttt{AddAdm}(uid, uid')$, $\mathcal{O}\texttt{RemoveAdm}(uid, uid')$**:** uid grants/revokes admin privileges from uid'. If successful, the oracle stores the control message in $\boldsymbol{T}[\mathsf{ep}]$, advances the global epoch $\mathsf{ep}{+}{+}$ and updates the set of group admins for the new epoch $\boldsymbol{AD}[\mathsf{ep}]$ by adding/removing uid'.

$\mathcal{O}\texttt{Update}(uid)$**:** can be called for any user uid (incl. non-admins) and updates the state of the caller. It advances the global epoch $\mathsf{ep}{+}{+}$ and the update counter $\mathsf{up}{+}{+}$.

$\mathcal{O}\texttt{RotKeys}(uid)$, $\mathcal{O}\texttt{UpdateAdm}(uid)$**:** uid rotates the group keys respectively refreshes its local admin state. If successful, the oracle stores the control message in $\boldsymbol{T}[\mathsf{ep}]$, and advances the global epoch $\mathsf{ep}{+}{+}$.

Security notions and cleanness. The precise provable security of a GKP scheme is defined by the cleanness predicate C, which determines the epoch keys that can be challenged by the adversary. The challenger verifies C at the end of the game, relative to all queries made by the adversary, and aborts the game if the predicate fails. C rules out trivial attacks, such as when $\mathcal{A}$ exposes a member in some epoch e and then asks to test a key known by the member at exposure time. The epoch-based modeling of the game allows C to capture interval access control by allowing $\mathcal{A}$ to test epoch keys that were derived before a compromised member entered a group, and after they were removed. Further, post-compromise security is captured by letting $\mathcal{A}$ test epoch keys known to users that were exposed in previous epochs, but recovered (e.g, through group key rotation) before the challenge epoch. However, forward security in the broader sense is not captured, as our GKP syntax does not allow that users delete epoch keys on-demand. We discuss cleanness further in our instantiation of GKP in Section 6.

A limitation of our model is that the adversary can only deliver messages that are generated honestly by the users. We chose to focus on capturing the core confidentiality properties of GKP to reduce complexity, as was done for CGKA modeling [6, 50]. Secure GKP schemes can be extended easily to achieve security against active adversaries. Concretely, authenticated GKP can be built from GKP using digital signatures and a PKI (see [7, 17] for analogous transformations for CGKA). We discuss security against active adversaries further in Section 7.

6 Grappa: Group Key Progression for Persistent Access

We introduce our instantiation of group key progression, called Grappa, which relies on an interval scheme INT and a CGKA scheme (see Section 4 and Section 3.2.3). We leverage recent advances from the messaging community to support group management features, establish a secure channel for group members, and enjoy an efficient service provider. To benefit from the fast fanout mechanism of the CGKA delivery service, we need to avoid sending user-specific messages over expensive pairwise channels. In particular, this implies that INT must have interval-agnostic extensions such that the same message can be sent to all group members. Finally, we need a private channel between group admins to allow them to synchronize on global state updates. Taking inspiration from the dual-CGKA protocol from [17], we instantiate such a channel via a separate (admin-exclusive) CGKA protocol.

Table 5: Summary of variables used throughout Grappa description.

$st_U.uid$	User's unique identifier
$st_U.\gamma_A$	User's admin CGKA state
$st_U.\gamma_M$	User's member CGKA state
$st_U.I_A$	User's admin CGKA group key
$st_U.I_M$	User's member CGKA group key
$st_U.e$	User's current epoch
$st_U.st$	User's interval state
$st_U.aux$	User's interval auxiliary value
ω	GKP welcome message
τ	GKP control message
ext	Interval extension
st	Interval state
fl	Progress flag, $fl \in \{\varepsilon, \leftarrow\!\shortmid, \shortmid\!\rightarrow\}$

T	CGKA control message
T_M	Control message for all members
T_A	Control message only for admins
W	CGKA welcome message
P	CGKA proposal message
cmd	CGKA or GKP command
I_A	Admin CGKA group key
I_M	Member CGKA group key
C_A	Ciphertext for admins, encrypted under I_A
C_M	Ciphertext for members, under I_M
C_ω	Ciphertext for joiners, under I_M

6.1 Grappa Construction

Below, we describe the algorithms of Grappa following the group key progression syntax: Grappa = (InitUser, Create, ExecCtrl, ProcCtrl, JoinCtrl, GetEpochKey). We provide the full specification in Figures 11 and 12 below, aided by a description in Table 5 of the variables involved in the pseudocode. Next, we describe the algorithms at a high level.

Group creation. A user *uid* can create a group with Create where they are admin. This operation creates two CGKA groups with only the calling user *uid* in them. It stores the CGKA member and admin states in γ_M and γ_A, respectively. Additionally, it initializes an interval state *st*.

Command execution. Group administrators can execute GKP commands *cmd* via ExecCtrl and output a control message τ and, only for command `Add`, a welcome message ω. For all commands *cmd*, the procedure is as follows, where steps 2–5 are only for admin commands (i.e., not `UpdUser`).

1. Generate a CGKA proposal message P on the appropriate command, followed by a commit message Commit and a fresh CGKA group key $st_U.I_M$ on the member CGKA.

2. For changes to the admin state (i.e., for $cmd \in \{$`AddAdm`, `RemAdm`, `UpdAdm`, `RotKeys`$\}$), additionally generate a CGKA proposal, commit message, and fresh group key on the admin CGKA. For additional security, refresh the member CGKA with an update (CGKA command `Upd`) even if that group composition does not change. This ensures that both CGKAs remain in-sync with respect to PCS updates, and is done by default in the helper algorithm RunBothCGKA.

3. Run INT.Progress to evolve the interval scheme, where the flag *fl* is selected according to *cmd* as described below. Also, derive an interval extension *ext* and, if necessary, export a new interval state *st* for added users/admins. This is done in the helper algorithm RunINT.

4. Encrypt the extension *ext* that parties require to keep their state in-sync, under the fresh member CGKA key $st_U.I_M$. Encryption is done using a symmetric AEAD encryption scheme AEAD, see e.g. Line 4 in Figure 12.

5. When a fresh backwards chain was picked (due to an explicit flag *fl* or exhausted chain), then encrypt the new auxiliary information *aux* for other admins, i.e., using AEAD on the admin key $st_U.I_A$ as in Line 56 in Figure 11.

6. Output a control message τ as a bundle of all ciphertexts and CGKA control messages.

InitUser(uid):

$\gamma_M \leftarrow\!\!\$$ CGKA.Init(uid); $\gamma_A \leftarrow\!\!\$$ CGKA.Init(uid)
$st_U.(uid, \gamma_M, \gamma_A) \leftarrow (uid, \gamma_M, \gamma_A)$
return st_U

Create(st_U):

// Create user and admin CGKA groups
$\gamma_M \leftarrow\!\!\$$ CGKA.Create($st_U.\gamma_M$)
$\gamma_A \leftarrow\!\!\$$ CGKA.Create($st_U.\gamma_A$)
$st \leftarrow$ INT.Init()
$st_U.(\gamma_M, \gamma_A, e, st) \leftarrow (\gamma_M, \gamma_A, 0, st)$
return st_U

JoinCtrl($st_U, \omega = (W, C_\omega)$):

$st_U.\gamma_M \leftarrow$ CGKA.Proc($st_U.\gamma_M, W$)
$st_U.st \leftarrow$ AEAD.Dec($st_U.\gamma_M.I, C_\omega$)
return st_U

ProcCtrl(st_U, τ):

pre: $\tau.e = st_U.e + 1$
$(cmd, T_A, T_M, W, C_A, C_M, e) \leftarrow \tau$
$st_U.\gamma_M \leftarrow$ CGKA.Proc($st_U.\gamma_M, T_M$)
if $cmd =$ `UpdUser`: **return** st_U
else: $++st_U.e$
if $st_U.uid \notin st_U.\gamma_M.G$: // if removed
  return InitUser($st_U.uid$) // erase state
if $cmd = (\texttt{AddAdm}, st_U.uid)$:
  // new admin receives full state and aux
  $(C_{st}, C_A) \leftarrow C_A$
$st_U \leftarrow$ ProcCtrlMember(st_U, τ, C_{st})
if $st_U.uid \in st_U.\gamma_A.G$:
  $st_U \leftarrow$ ProcCtrlAdmin(st_U, τ, C_A)
return st_U

GetEpochKey(st_U, e):

return INT.GetKey($st_U.st, e$)

ProcCtrlAdmin(st_U, τ, C_A):

$(cmd, T_A, \cdot, \cdot, \cdot, C_M, \cdot) \leftarrow \tau$
$st_U.\gamma_A \leftarrow$ CGKA.Proc($st_U.\gamma_A, T_A$)
// if admin rights revoked
if $st_U.uid \notin st_U.\gamma_A.G$:
  $st_U.\gamma_A \leftarrow$ CGKA.Init($st_U.uid$)
  $ext \leftarrow$ AEAD.Dec($st_U.\gamma_M.I, C_M$)
  // process extension as normal member
  $st_U.st \leftarrow$ INT.ProcExt($st_U.st, ext$)
  return st_U
$st_U.aux \leftarrow$ AEAD.Dec($st_U.\gamma_A.I, C_A$)
return st_U

ProcCtrlMember(st_U, τ, C_{st}):

$(cmd, \cdot, \cdot, W, \cdot, C_M, \cdot) \leftarrow \tau$
if $cmd = (\texttt{AddAdm}, st_U.uid)$:
  // new admin: join group and receive admin state
  $st_U.\gamma_A \leftarrow$ CGKA.Join($st_U.\gamma_A, W$)
  $st_U.st \leftarrow$ AEAD.Dec($st_U.\gamma_A.I, C_{st}$)
else: // process interval extensions
  $ext \leftarrow$ AEAD.Dec($st_U.\gamma_M.I, C_M$)
  $st_U.st \leftarrow$ INT.ProcExt($st_U.st, ext$)
return st_U

RunCGKA(γ, cmd, uid):

$(\gamma, P) \leftarrow\!\!\$$ CGKA.Prop(γ, cmd, uid)
$(\gamma, T, W, I) \leftarrow\!\!\$$ CGKA.Commit(γ, P)
return (γ, T, W, I)

RunBothCGKA(st_U, cmd, uid):

$(st_U.\gamma_A, T_A, W, st_U.I_A) \leftarrow$ RunCGKA($st_U.\gamma_A, cmd, uid$)
$(st_U.\gamma_M, T_M, \perp, st_U.I_M) \leftarrow$ RunCGKA($st_U.\gamma_M, \texttt{Upd}, \perp$)
return (st_U, T_A, T_M, W)

RunINT(st_U, fl):

$st_U.e \leftarrow st_U.e + 1$
$(st_U.st, aux) \leftarrow\!\!\$$ INT.Progress($st_U.st, st_U.aux, fl$)
if $aux \neq \varepsilon$:
  $st_U.aux \leftarrow aux$; $C_A \leftarrow$ AEAD.Enc($st_U.I_A, aux$, 'INTAux')
else: $C_A \leftarrow \varepsilon$
$ext \leftarrow\!\!\$$ INT.CreateExt($st_U.st, st.e_{max} - 1, 1$)
$C_M \leftarrow$ AEAD.Enc($st_U.I_M, ext$, 'INTExt')
return (st_U, C_A, C_M)

Figure 11: GKP instantiation Grappa with helper functions in the bottom two boxes of the figure. The ExecCtrl algorithm is in Figure 12.

$\underline{\mathsf{ExecCtrl}(st_U, \texttt{Add}, uid)}$:

$(st_U.\gamma_M, T_M, W, st_U.I_M) \leftarrow \mathsf{RunCGKA}(st_U.\gamma_M, \texttt{Add}, uid)$
$(st_U, C_A, C_M) \leftarrow \mathsf{RunINT}(st_U, \varepsilon)$
$st' \leftarrow \mathsf{INT.Export}(st_U.st, st_U.e_{max}, st_U.e_{max})$
$C_\omega \leftarrow \mathsf{AEAD.Enc}(st_U.I_M, st', \text{'INTUserState'})$
return $(st_U, \tau = (\texttt{Add}, \varepsilon, T_M, \varepsilon, C_A, C_M, st_U.e), \omega = (W, C_\omega))$

$\underline{\mathsf{ExecCtrl}(st_U, \texttt{Rem}, uid)}$:

$(st_U.\gamma_M, T_M, \cdot, st_U.I_M) \leftarrow \mathsf{RunCGKA}(st_U.\gamma_M, \texttt{Rem}, uid)$
$(st_U, C_A, C_M) \leftarrow \mathsf{RunINT}(st_U, \mapsto)$
return $(st_U, \tau = (\texttt{Rem}, \varepsilon, T_M, \varepsilon, C_A, C_M, st_U.e), \bot)$

$\underline{\mathsf{ExecCtrl}(st_U, \texttt{AddAdm}, uid)}$:

$(st_U, T_A, T_M, W) \leftarrow \mathsf{RunBothCGKA}(st_U, \texttt{AddAdm}, uid)$
$(st_U, C_A', C_M) \leftarrow \mathsf{RunINT}(st_U, \varepsilon)$
// Encrypt admin state for new admin (in addition to aux already encrypted in C_A' for all existing admins).
$C_A \leftarrow (\mathsf{AEAD.Enc}(I_A, st_U.st, \text{'INTAdminState'}), C_A')$
return $(st_U, \tau = ((\texttt{AddAdm}, uid), T_A, T_M, W, C_A, C_M, st_U.e), \bot)$

$\underline{\mathsf{ExecCtrl}(st_U, \texttt{RemAdm}, uid)}$:

$(st_U, T_A, T_M, \cdot) \leftarrow \mathsf{RunBothCGKA}(st_U, \texttt{Rem}, uid)$
$(st_U, C_A, C_M) \leftarrow \mathsf{RunINT}(st_U, \leftarrow\!\shortmid)$
return $(st_U, \tau = (\texttt{RemAdm}, T_A, T_M, \varepsilon, C_A, C_M, st_U.e), \bot)$

$\underline{\mathsf{ExecCtrl}(st_U, \texttt{UpdAdm}, uid)}$:

$(st_U, T_A, T_M, \cdot) \leftarrow \mathsf{RunBothCGKA}(st_U, \texttt{Upd}, \bot)$
$(st_U, C_A, C_M) \leftarrow \mathsf{RunINT}(st_U, \varepsilon)$
return $(st_U, \tau = (\texttt{AddAdm}, T_A, T_M, \varepsilon, C_A, C_M, st_U.e), \bot)$

$\underline{\mathsf{ExecCtrl}(st_U, \texttt{RotKeys}, uid)}$:

$(st_U, T_A, T_M, \cdot) \leftarrow \mathsf{RunBothCGKA}(st_U, \texttt{Upd}, \bot)$
$(st_U, C_A, C_M) \leftarrow \mathsf{RunINT}(st_U, \leftarrow\!\shortmid)$
return $(st_U, \tau = (\texttt{RotKeys}, T_A, T_M, \varepsilon, C_A, C_M, st_U.e), \bot)$

$\underline{\mathsf{ExecCtrl}(st_U, \texttt{UpdUser}, uid)}$:

$(st_U.\gamma_M, T_M, st_U.I_M) \leftarrow \mathsf{RunCGKA}(st_U.\gamma_M, \texttt{Upd}, \bot)$
return $(st_U, \tau = (\texttt{UpdUser}, \varepsilon, T_M, \varepsilon, \varepsilon, \varepsilon, st_U.e), \bot)$

Figure 12: Remaining ExecCtrl commands for the GKP instantiation Grappa.

The commands *cmd* set in ExecCtrl have the following effect:

`Add:` User *uid* is added to the member CGKA group, producing CGKA welcome message W. No progress flag is set, but members still use a new forward chain element because the epoch is advanced. The admin exports an interval state st' for the current epoch and sends it to user *uid* encrypted under the member CGKA key $st_U.I_M$ as part of the GKP welcome message ω (together with CGKA welcome message W).

`Rem:` User *uid* is removed from the member CGKA and the interval is progressed with flag $\mapsto$, creating a new forward chain so *uid* cannot derive future keys.

`AddAdm:` User *uid* is added to the admin CGKA group and progresses the interval with the empty flag as all admins get access to the global all keys.

`RemAdm:` Admin *uid* is removed from the admin CGKA and progress runs with flag $\leftarrow\!\shortmid$ to create a new backward chain (stored in *aux* and sent to other admins), as the revoked admin knows the previous backward chain seed.

`UpdAdm:` Update both member and admin CGKAs by proposing and committing an `Upd` CGKA command, and progress the interval with the empty flag.

`RotKeys:` Update both member and admin CGKAs and progress the interval with flag $\leftarrow\!\shortmid$ to create a new backward chain.

`UpdUser:` Refresh the local CGKA state by proposing and committing an update `Upd` in the member CGKA. This is the only operation that does not progress the interval and *can be called by any member* (including non-admins).

Command processing. The protocol ProcCtrl processes a GKP control message τ. For all commands, this message contains a member CGKA control message that is processed first. Command `UpdUser` terminates here (Line 15 of Figure 11), all others increment the GKP epoch. If the executed command was `Rem`, Grappa erases that user's local state when it calls ProcCtrl (Line 18). For all other commands except `AddAdm`, members move to the next epoch key by processing the interval extension, which is encrypted with the member CGKA key. For `AddAdm`, the new admin instead joins the admin CGKA by processing the CGKA welcome message and then uses the admin CGKA key to decrypt the global interval state (Line 20 of Figure 11). Admins additionally need to process control messages for the admin CGKA and decrypt new auxiliary information from other admins (which, if necessary, contains a new backwards chain seed encrypted with the admin CGKA key). The only exception is if the user's admin rights were revoked with `RemAdm`, in which case they will not learn the new seed and need to extend the interval by processing the extension as all non-admins. Current admin membership is checked using the user's state itself (see Line 23).

Joining the group. A joining user receives a GKP welcome message ω, which is processed in the JoinCtrl algorithm. This contains a CGKA welcome message W that the user processes to obtain the member CGKA key. This key allows the user to decrypt their interval state.

Obtaining epoch keys. GKP epoch keys are obtained directly from the underlying interval scheme whose epochs are synchronized with GKP epochs. This is done in algorithm GetEpochKey.

6.2 Grappa Security

The following theorem shows that Grappa is secure with respect to the security notion for GKP schemes in Section 5.2. The exact security of the construction is parametrized by the predicate C, that we present in Appendix F due to space constraints. The predicate may generically depend on the cleanness predicate $\mathsf{C}_{\mathsf{CGKA}}$ of CGKA. For the proof, we adopt the optimally-secure safety predicate $\mathsf{C}_{\mathsf{CGKA}}$ from [6], achieved by their RTreeKEM construction. We also take the safety predicate from our interval scheme $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ in Figure 8, and therefore state security with respect to this construction. Note, however, that security holds for any of the interval-agnostic interval schemes discussed in Section 4.6 with minor tweaks on the predicate C. We provide a proof outline at the end of this section and the full proof in Appendix F.

Theorem 6.1 (Key indistinguishability of Grappa)**.** Let CGKA be a CGKA scheme with group key length ℓ_a, let AEAD be an AEAD scheme with key length ℓ_a and let INT be our $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ scheme. Let Grappa be the group key progression scheme defined from CGKA, AEAD and INT as per Figures 11 and 12. Let $\mathcal{A}$ be a non-adaptive adversary against GKP key indistinguishability (Definition 5.2) with respect to cleanness predicate C that makes Q oracle queries. Then we can construct adversaries $\mathcal{B}_{\mathsf{CGKA}}$, $\mathcal{B}_{\mathsf{INT}}$, $\mathcal{B}_{\mathsf{AEAD}}$ such that

$$\begin{aligned}\mathbf{Adv}^{\text{kind}}_{\mathsf{Grappa},\mathsf{C}}(\mathcal{A}) \leq\ & 2Q \cdot \mathbf{Adv}^{\text{kind}}_{\mathsf{CGKA},\mathsf{C}_{\mathsf{CGKA}}}(\mathcal{B}_{\mathsf{CGKA}}) \\ & + \mathbf{Adv}^{\text{iac}}_{\mathsf{INT}}(\mathcal{B}_{\mathsf{INT}}) + 2 \cdot \mathbf{Adv}^{\text{ind\$-cpa}}_{\mathsf{AEAD}}(\mathcal{B}_{\mathsf{AEAD}})\ .\end{aligned}$$

Adversary $\mathcal{B}_{\mathsf{CGKA}}$ makes at most $3Q$ queries, $\mathcal{B}_{\mathsf{AEAD}}$ makes at most Q queries, and $\mathcal{B}_{\mathsf{INT}}$ makes at most Q queries to the oracles of their respective challengers. The running times of all adversaries are approximately that of $\mathcal{A}$.

In Grappa, the IAC security of the interval scheme constrains the access of users to an interval of keys. To remain secure against multiple user compromises, Grappa has to carefully set the appropriate flags to create new chains on `Rem`, `RemAdm`, and `RotKeys` commands. Both CGKAs allow members and admins to share interval state extensions via a channel that provides FS and PCS. Note that Grappa also achieves PCS for keys, but only due to the combination of INT and CGKA: regular updates to a user's CGKA state ensure that $\mathcal{A}$ does not learn the interval extension required to compute future epoch keys. Recovering from an admin compromise additionally requires running `RotKeys` (e.g., to move to a new backward chain unknown to $\mathcal{A}$). The number of group admins in Grappa has security implications. Having a large pool of admins increases the attack surface, as an admin compromise reveals more cryptographic material than a standard member compromise. However, a small and frequently unavailable group of admins may degrade security as they are needed to progress the interval scheme, gain PCS for members, and execute membership commands.

Theorem 6.1 proof outline. In the first step of the proof, we define the cleanness predicate $\mathsf{C} = \mathsf{C}(e^*)$ with respect to the challenge epoch e^*, which corresponds to the epoch key that was tested by the adversary via $\mathcal{O}\texttt{Test}$. The predicate C allows the adversary to test epoch keys from any epoch for which none of the exposed users was a group member, capturing IAC. It also captures PCS by allowing the corruptions of parties *uid* that occur before they heal via $\mathcal{O}\texttt{RotKeys}(\mathit{uid})$ (or, in some cases, also via $\mathcal{O}\texttt{Update}(\mathit{uid})$).

The proof then proceeds via a series of game hops, where Game 0 is the original GKP key indistinguishability game. First, we consider the exposures of group administrators, and define an epoch e_{AD} as the earliest epoch for which all admins have healed (after being exposed in the past). Note that $\mathsf{C}(e) = \textbf{false}$ if $e^* < e_{\mathsf{AD}}$. Then, we define Game 1 as Game 0 except that all CGKA

admin keys corresponding to epochs $e \geq e_{\mathsf{AD}}$ are replaced by random keys. This step follows via a reduction to key indistinguishability of the admin CGKA, which can be carried out as all the participants (GKP admins) have healed at e_{AD}.

Second, we define Game 2 as Game 1 except that all ciphertexts encrypted under admin CGKA-derived keys after e_{AD} are replaced by random. The step easily follows by AEAD security as all the admin CGKA keys were already random.

Third, we consider the exposures of any user and capture a so-called safety window such that (1) e^* is in the safety window, and (2) the cleanness predicate is true for all GKP epochs in the safety window. We define Game 3 as Game 2 except that we replace all member CGKA keys in the safety window by random. The reduction follows as in Game 1, by showing first that the CGKA cleanness predicate is valid for all the corresponding CGKA epochs.

Fourth, we define Game 4 as Game 3 except that we replace ciphertexts encrypted with member CGKA keys for epochs in the safety window by random, as we did in Game 2.

Finally, we consider the effect of user exposures in the safety predicate Safe of $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$. We describe a so-called flag window, which is contained in the safety window, which considers the effects of progress flags (i.e., the creation of new chains) with respect to user exposures. We also show that, unless e^* is in a non-empty flag window, then $\mathsf{C}(e^*) = \textbf{false}$. We define Game 5 as Game 4 except that all epoch keys of $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ in the flag window (except for those queried via $\mathcal{O}\mathtt{GetKey}$) are replaced by random. The reduction to IAC security of $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ follows by a standard simulation strategy, where GKP keys exposed via $\mathcal{O}\mathtt{GetKey}$ are obtained via the respective IAC oracle $\mathcal{O}\mathtt{GetKey}$. The proof concludes as the epoch key corresponding to e^* is random, hence the advantage of the adversary in Game 5 is zero. The full proof is in Appendix F.

6.3 Grappa Example Showing Interaction with Interval Scheme $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$

Figure 13 provides an example that visualizes how Grappa, when instantiated with the DKR-based interval scheme from one-way chains $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$, generates a group key progression leveraging the interval scheme. In this example, a user U is a group member from epoch e_i to e_j, and hence knows the chain elements necessary to derive all forward and backward chain elements for that interval. This includes multiple chains as Grappa replaced both forward and backward chains when admin A left the group in epoch e^*. From the chain elements, user U can derive the epoch keys k_e for $e \in [i, j]$. However, user U cannot compute keys for $x \notin [i, j]$: for epoch keys before e_i, the forward chain elements are unknown, and, similarly, after epoch e_j the backward chain elements are missing. In fact, Grappa provides even stronger security by rotating the forward chain when user U is removed from the group, which prevents an adversary who knows a later backward seed from deriving intermediate values.

6.4 Efficiency of Grappa

This section compares the storage and communication cost of Grappa when instantiated with all interval schemes (for $\ell = 128$-bit keys) introduced in Section 4 that have interval-agnostic extensions: $\mathsf{trivial}$, $\mathsf{GGM1}$, $\mathsf{D}[\mathsf{S}, \mathsf{F}]$, and $\mathsf{D}[\mathsf{H}, \mathsf{F}]$.

The concrete storage and communication costs depend on the sequence of GKP operations. Hence, we implement and publish[17] a simulator for Grappa that supports the above interval schemes and tracks the state and message sizes (but not the CGKA overhead which is independent from the interval scheme). In Table 6, we report the results of a system starting with 500 users and 5 admins, averaged over 100 runs of sampling 10^5 operations at random from a distribution of operations

[17]The simulator is published here: https://github.com/Miro-H/GrappaSimulator.

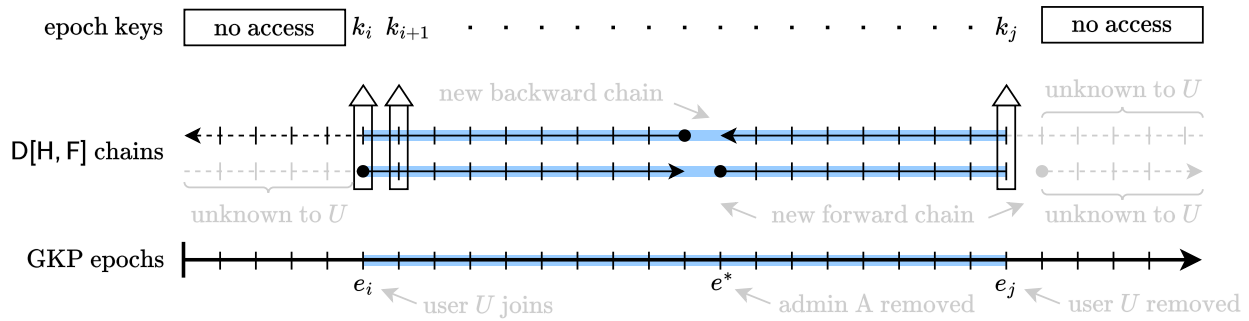

Figure 13: Visualizing IAC for user U in GKP instantiation Grappa and the interaction with the forward and backward one-way chains of the internal building block D[OWC, F]. User U, joining in epoch e_i and leaving the group in epoch e_j has only access to keys $k_i, k_{i+1}, \ldots, k_j$. While part of the group, user U may learn multiple chains, e.g., when admin A is removed in epoch e^* and both chains are rotated.

Table 6: Simulated storage costs for Grappa using different interval schemes, averaged over 100 simulations each sampling 10^5 operations with the following probabilities: 1% `Add`, 1% `Rem`, 48.9895% `UpdUser`, 0.01% `AddAdm`, 0.01% `RemAdm`, 48.9895% `UpdAdm`, and 0.001% `RotKeys`. The simulation starts with 500 users, 5 admins, and uses a maximum chain length $N' = 10^4$ producing $\ell = 128$-bit keys. The communication complexity is the sum of all exchanged admin/user messages. We highlight the best cells per column.

	state size		**avg message size**		**com. complexity**	
	admin	*user*	*admin*	*user*	*admin*	*user*
trivial	816 KB	272 KB	40.5 B	16.0 B	7.85 MB	406 MB
GGM1	38.4 KB	269 KB	233 B	16.0 B	203 KB	406 MB
D[S, F]	5.34 KB	6.67 KB	821 B	119 B	26.3 KB	3.07 GB
D[H, F]	4.2 KB	5.5 KB	610 B	16.3 B	20.8 KB	414 MB

(detailed in the caption of Table 6) that we consider realistic for envisioned applications. In our distribution, `UpdUser` and `UpdAdm` are by far the most common operations as their regular execution provides PCS. Adding/removing members is one order of magnitude more common than changing the set of admins. Finally, using `RotKeys` to manually rotate all admin key material without changing the group composition is a rare operation. To gain better intuition for our distribution consider the following example. If 10^5 Grappa operations are executed in a year, then we have roughly 5.6 executions each for `UpdUser` and `UpdAdm` per hour, every week there are 19.6 users added and removed, and over the entire year 10 admins are added and removed, and we have 1 `RotKeys` operation. We encourage practitioners to adapt the operation probabilities to their use case and run our simulator to evaluate the best configuration.

In Table 6, the admin state of trivial is larger than user states because, by definition, they include the information to derive all epoch keys and not only a subset. The other schemes can take advantage of compactness, from which admins tend to profit most (e.g., in GGM1, admins only store the tree roots). User messages are only a single 128-bit key, tree leaf, or hash-chain element for trivial, GGM1, and D[H, F], respectively. The exception is SSKG, where chain states consist of paths in a tree that depend on the start of the chain. The total communication complexity of users is larger than the one of admins because there are much more users than admins in our simulation.

In summary, we conclude that our Grappa scheme with D[H, F] has the most compact states

because they do *not* grow with the number of epochs passed since a user joined the group, but instead only grow linearly with the comparatively small number of calls to `Rem`, `RemAdm`, and `RotKeys` (or when the chain is exhausted). Although the message size is slightly larger on average, $\mathsf{D}[\mathsf{H}, \mathsf{F}]$ has less overall communication between admins due to two factors: 1) DKR-based schemes can often avoid sending admin messages unless *aux* changed and 2) Grappa states are more compact and thus transferring full states to new users or admins is less costly.

7 Discussion

We introduced *group key progression*, a primitive that allows a dynamic group of users with administrators and members to derive keys for persistent data. These keys satisfy interval access control, meaning that a user cannot derive keys from before it was granted access and after it was removed. In addition to provable security, our constructions target practical efficiency. The remainder of this section outlines applications of Grappa (and more generally GKP), directions for future research, and a series of possible extensions to GKP and Grappa.

7.1 Applications

Figure 14 visualizes a generic application that uses Grappa to establish a progression of common keys in a group. These keys can then be input to a key schedule $\mathcal{K}$ to derive or encrypt keys used to secure persistent data. For example, Grappa could be used to build a shared folder, where one or more folder owners are GKP admins that may add and remove other users. Every user can derive epoch keys—in the cloud storage context often called key encryption keys (KEKs)—to encrypt uniformly random and freshly sampled file keys, which in turn are used to encrypt files. KEKs enable further application-level functionality such as file sharing (see [12]) or data key rotation. Upon user compromise, IAC ensures that files uploaded before or after a user was granted access remain secure, and hence limit the impact of a compromise even if the adversary gets ciphertext access. Moreover, PCS allows Grappa to lock out passive adversaries by regularly refreshing key material.

Other applications include backups for end-to-end encrypted (E2EE) messages, where a group of users can use Grappa to encrypt exchanged messages (see Appendix C). Our security notions make explicit how this transition from data in transit to persistent data at rest changes the provided guarantees, including FS and PCS. Furthermore, Grappa could be used to store E2EE recordings of video calls. Epoch keys together with appropriate slicing of video recordings could ensure that key material is rotated as participants join and leave a meeting while still granting everyone access to the recording of the part they attended.

7.2 Future Work

As discussed in previous sections, the GKP security model does not capture adversaries that can inject messages in the network. Hence, security mechanisms against active attacks or malicious insiders, such as signing control messages, cannot be captured in our model. Our model can be extended to support active adversaries by adding another argument to $\mathcal{O}\texttt{Deliver}$ and $\mathcal{O}\texttt{Join}$ such that $\mathcal{A}$ can deliver arbitrary messages, instead of only those in $\boldsymbol{T}$ and $\boldsymbol{W}$.

Beyond authentication security, which is relatively straightforward by using signatures and a PKI, the ultimate goal would be to build an efficient GKP that is fully secure against both malicious servers and insiders. Unfortunately, this is challenging for the group key agreement component of GKP, as current CGKAs with full active security [8, 34] are impractically expensive. However, we

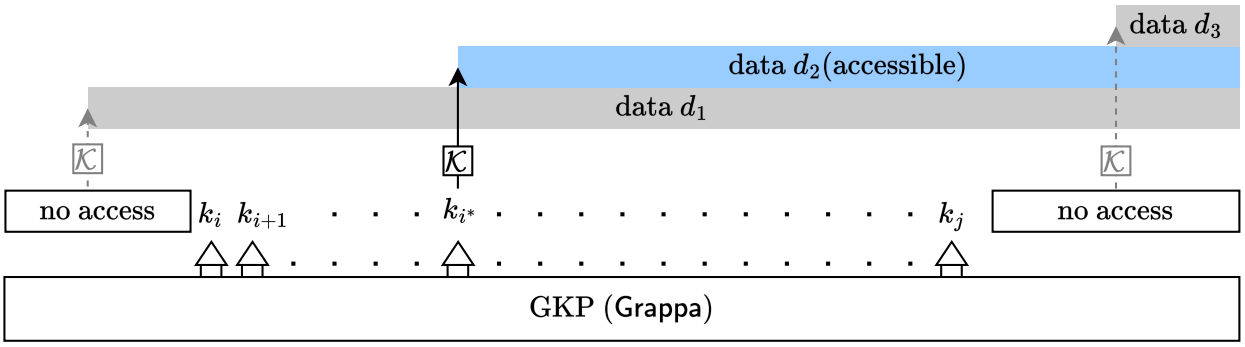

Figure 14: Example application built on top of Grappa, where the derivable keys e_i are input to a key schedule $\mathcal{K}$ to derive keys to protect persistent data.

remark that a trusted delivery service for GKP does not prevent a system from achieving malicious security for stored data (e.g., for confidentiality and integrity as in [12]) because it could have a separate storage server with weaker trust assumptions. While it is challenging to *prevent* active attacks, in practice we might be satisfied with *detecting* such attacks. A promising direction is to analyze whether active attack detection mechanisms for two-party messaging [19] can be extended to groups. Another open question is to prove the adaptive security of Grappa, as our proofs only show selective security.

Alternative constructions for GKP are another interesting direction because, for example, CGKA may be considered a too heavy primitive for small group settings. GKP can be built from an interval scheme and a different group communication protocol such as pairwise channels or Sender Keys, offering varying PCS guarantees. In the case of pairwise channels, interval schemes with interval-dependent extensions such as GGM trees could also be used seamlessly.

7.3 GKP Extensions

Finally, several possible extensions of interval schemes and GKP include granting users history access to epoch keys before they joined, temporary access revocation, and efficiency improvements for batching command execution and resetting the growing state size.

Granting history access. Our GKP operation `Add` grants newly added users access from the current epoch e. Our primitive could be extended to allow the inviting user to specify an epoch $e' \leq e$ as an additional argument to `Add` to denote from when the added user can derive keys. In the current scheme, Line 4 of Figure 12 uses the current CGKA group key $st_U.I_M$ of epoch e to encrypt the key material for the new user. In our threat model, this encryption would leak the keys for epochs $[e', e-1]$ to other group members who may not be allowed to access keys before epoch $e-1$ but know $st_U.I_M$. Consequently, implementing history access would require access to secure user-to-user channels to distribute the key material for epochs $[e', e]$ exclusively to the invited user. In practice, one could leverage the PKI infrastructure that is already used by CGKA for identity keys. However, this opens up the CGKA abstraction and so we leave its formal analysis for future work. A more direct alternative is to use pairwise channels on top of interval schemes as discussed above.

Additional GKP and INT commands. Other commands that could be considered to add advanced functionality to GKP and interval schemes are the following:

- `TmpRevokeAccess`, `RestoreAccess`: Inspired by BurnBox [64], we could support that users can temporarily revoke their own access to the group (`TmpRevokeAccess`) and later restore it (`RestoreAccess`). This could be implemented e.g. with secret sharing.

- $\mathsf{INT.CreateExtLeft}(st, l, r, s)$: This algorithm is the analog to $\mathsf{INT.CreateExt}$ for creating interval scheme extensions, but for past instead of future epochs: it extends the access of an interval state covering epochs $[l, r]$ to $[l - s, r]$, i.e., to all keys up to epoch $l - s$. Similar to granting history access, this command requires secure pairwise channels.

- $\mathsf{INT.Merge}(st_1, st_2)$: We could add an algorithm to merge two interval states st_1 and st_2 to cover the union of their intervals. This could be allowed only for overlapping states to increase efficiency of applications where the same user may receive multiple interval states. Alternatively, we could allow an interval state to cover multiple intervals, but this would come at the cost of complicating our syntax, construction, and security notion.

Efficiency improvements. Currently all control commands and operations below are performed for individual users. Similar to CGKA, we could allow administrators to badge multiple commands together. While this makes their processing faster, one needs to consider that the security benefit of operations is deferred until after the batched execution, and, hence, does not apply to commands running in the same batch.

Efficiency can be improved by enabling a "key sequence reset" operation in interval schemes, which replaces the entire key sequence with a new sequence that ignores all previously set progress flags. This would require the application to re-encrypt ciphertexts protected with GKP keys (e.g., in a shared folder for E2EE data where GKP keys encrypt file keys, these file keys would need to be re-encrypted by admins). Every active interval (e.g., a user in Grappa) would then receive the new interval state that is necessary to derive all keys of their access period. This would allow an application to reset the state growth. However, this operation trades off security, as the key sequence loses all intermediate progress flags and thus might leaks keys when multiple users are compromised for interval schemes with relaxed IAC guarantees (cf. our DKR-based instantiation from Section 4.4). Nonetheless, GKP states from before the reset still provide the same IAC guarantees for their GKP keys and the corresponding old application ciphertexts as before the reset, i.e., their compromise does not leak additional keys. Depending on the application using the interval scheme, such an extension and performance-security trade-off may be beneficial.

Automatic access expiration. Finally, GKP can be instantiated with a time-based key schedule, which progresses the global interval at regular time steps (say, every 10 minutes) as well as on group membership changes. Thus, an application using GKP could automatically expire a user's access at some date (or after a certain number of epochs). This could be taken even further such that the user's data is also deleted after their access expired. Automatic, time-based self-destruction of data has been studied in the past, see e.g. [40], but these constructions are not easily compatible with our setting. Hence, it is an open question how to achieve self-destructing data for GKP.

Acknowledgments. The research of Miro Haller was supported by an Amazon Research Award Fall 2023. Any opinions, findings, and conclusions or recommendations expressed in this material are those of the author and do not reflect the views of Amazon.

The research of David Balbás is supported by the PICOCRYPT project that has received funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (Grant agreement No. 101001283), partially supported by

projects PRODIGY (TED2021-132464B-I00) and ESPADA (PID2022-142290OB-I00) funded by MCIN/AEI/10.13039/501100011033/ and the European Union NextGenerationEU / PRTR, and partially supported by Ministerio de Universidades (FPU21/00600).

We thank Mihir Bellare for his contributions to the proof of Lemma 4.5 on the relations between dual- and double-PRF security. We also thank Nicola Dardanis and Matteo Scarlata for insightful discussions and comments during the elaboration of the draft, as well as Felix Günther, Kenny Paterson, Keegan Ryan, and Laura Shea for their helpful feedback.

References

[1] Martin R. Albrecht, Matilda Backendal, Daniele Coppola, and Kenneth G. Paterson. Share with care: Breaking E2EE in nextcloud. In *2024 IEEE European Symposium on Security and Privacy*, pages 828–840. IEEE Computer Society Press, July 2024.

[2] Martin R. Albrecht, Benjamin Dowling, and Daniel Jones. Device-oriented group messaging: A formal cryptographic analysis of matrix' core. In *2024 IEEE Symposium on Security and Privacy*, pages 2666–1685. IEEE Computer Society Press, May 2024.

[3] Martin R. Albrecht, Miro Haller, Lenka Mareková, and Kenneth G. Paterson. Caveat implementor! Key recovery attacks on MEGA. In Carmit Hazay and Martijn Stam, editors, *EUROCRYPT 2023, Part V*, volume 14008 of *LNCS*, pages 190–218. Springer, Cham, April 2023.

[4] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. CoCoA: Concurrent continuous group key agreement. In Orr Dunkelman and Stefan Dziembowski, editors, *EUROCRYPT 2022, Part II*, volume 13276 of *LNCS*, pages 815–844. Springer, Cham, May / June 2022.

[5] 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, Cham, May 2019.

[6] 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, Cham, August 2020.

[7] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Modular design of secure group messaging protocols and the security of MLS. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1463–1483. ACM Press, November 2021.

[8] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. Continuous group key agreement with active security. In Rafael Pass and Krzysztof Pietrzak, editors, *TCC 2020, Part II*, volume 12551 of *LNCS*, pages 261–290. Springer, Cham, November 2020.

[9] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. In Yevgeniy Dodis and Thomas Shrimpton, editors, *CRYPTO 2022, Part II*, volume 13508 of *LNCS*, pages 34–68. Springer, Cham, August 2022.

[10] Joël Alwen, Marta Mularczyk, and Yiannis Tselekounis. Fork-resilient continuous group key agreement. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 396–429. Springer, Cham, August 2023.

[11] Matilda Backendal, Mihir Bellare, Felix Günther, and Matteo Scarlata. When messages are keys: Is HMAC a dual-PRF? In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part III*, volume 14083 of *LNCS*, pages 661–693. Springer, Cham, August 2023.

[12] Matilda Backendal, Hannah Davis, Felix Günther, Miro Haller, and Kenneth G. Paterson. A formal treatment of end-to-end encrypted cloud storage. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part II*, volume 14921 of *LNCS*, pages 40–74. Springer, Cham, August 2024.

[13] Matilda Backendal, Felix Günther, and Kenneth G. Paterson. Puncturable key wrapping and its applications. In Shweta Agrawal and Dongdai Lin, editors, *ASIACRYPT 2022, Part II*, volume 13792 of *LNCS*, pages 651–681. Springer, Cham, December 2022.

[14] Matilda Backendal, Miro Haller, and Kenneth G. Paterson. MEGA: Malleable encryption goes awry. In *2023 IEEE Symposium on Security and Privacy*, pages 146–163. IEEE Computer Society Press, May 2023.

[15] Michael Backes, Christian Cachin, and Alina Oprea. Secure key-updating for lazy revocation. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *ESORICS 2006*, volume 4189 of *LNCS*, pages 327–346. Springer, Berlin, Heidelberg, September 2006.

[16] David Balbás, Daniel Collins, and Phillip Gajland. WhatsUpp with sender keys? Analysis, improvements and security proofs. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part V*, volume 14442 of *LNCS*, pages 307–341. Springer, Singapore, December 2023.

[17] David Balbás, Daniel Collins, and Serge Vaudenay. Cryptographic administration for secure group messaging. In Joseph A. Calandrino and Carmela Troncoso, editors, *USENIX Security 2023*, pages 1253–1270. USENIX Association, August 2023.

[18] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The messaging layer security (mls) protocol. RFC 9420, July 2023.

[19] Khashayar Barooti, Daniel Collins, Simone Colombo, Loïs Huguenin-Dumittan, and Serge Vaudenay. On active attack detection in messaging with immediate decryption. In Helena Handschuh and Anna Lysyanskaya, editors, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 362–395. Springer, Cham, August 2023.

[20] Mihir Bellare, Alexandra Boldyreva, and Silvio Micali. Public-key encryption in a multi-user setting: Security proofs and improvements. In Bart Preneel, editor, *EUROCRYPT 2000*, volume 1807 of *LNCS*, pages 259–274. Springer, Berlin, Heidelberg, May 2000.

[21] Mihir Bellare, Ran Canetti, and Hugo Krawczyk. Keying hash functions for message authentication. In Neal Koblitz, editor, *CRYPTO'96*, volume 1109 of *LNCS*, pages 1–15. Springer, Berlin, Heidelberg, August 1996.

[22] Mihir Bellare and Anna Lysyanskaya. Symmetric and dual PRFs from standard assumptions: A generic validation of an HMAC assumption. Cryptology ePrint Archive, Report 2015/1198, 2015.

[23] Mihir Bellare and Phillip Rogaway. The security of triple encryption and a framework for code-based game-playing proofs. In Serge Vaudenay, editor, *EUROCRYPT 2006*, volume 4004 of *LNCS*, pages 409–426. Springer, Berlin, Heidelberg, May / June 2006.

[24] Alysson Bessani, Miguel Correia, Bruno Quaresma, Fernando André, and Paulo Sousa. Depsky: Dependable and secure storage in a cloud-of-clouds. *ACM Trans. Storage*, 9(4), nov 2013.

[25] 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, Berlin, Heidelberg, August 2013.

[26] 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, Berlin, Heidelberg, December 2013.

[27] 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, Cham, August 2020.

[28] 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, Berlin, Heidelberg, March 2014.

[29] Lukas Burkhalter, Anwar Hithnawi, Alexander Viand, Hossein Shafagh, and Sylvia Ratnasamy. TimeCrypt: Encrypted data stream processing at scale with cryptographic access control. In *17th USENIX Symposium on Networked Systems Design and Implementation (NSDI 20)*, pages 835–850, Santa Clara, CA, February 2020. USENIX Association.

[30] Dajiang Chen, Li Zhang, Zeyu Liao, Hong-Ning Dai, Ning Zhang, Xuemin Shen, and Minghui Pang. Flexible and fine-grained access control for EHR in blockchain-assisted e-healthcare systems. *IEEE Internet of Things Journal*, 2023.

[31] Long Chen, Hui Guo, Ya-Nan Li, and Qiang Tang. Efficient secure storage with version control and key rotation. In Jian Guo and Ron Steinfeld, editors, *ASIACRYPT 2023, Part VI*, volume 14443 of *LNCS*, pages 168–198. Springer, Singapore, December 2023.

[32] Weikeng Chen, Thang Hoang, Jorge Guajardo, and Attila A. Yavuz. Titanium: A metadata-hiding file-sharing system with malicious security. In *NDSS 2022*. The Internet Society, April 2022.

[33] Gareth T. Davies, Sebastian H. 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, *CRYPTO 2023, Part IV*, volume 14084 of *LNCS*, pages 330–361. Springer, Cham, August 2023.

[34] Julien Devigne, Céline Duguey, and Pierre-Alain Fouque. MLS group messaging: How zero-knowledge can secure updates. In Elisa Bertino, Haya Shulman, and Michael Waidner, editors, *ESORICS 2021, Part II*, volume 12973 of *LNCS*, pages 587–607. Springer, Cham, October 2021.

[35] Yevgeniy Dodis and Daniel Jost. Compact key storage in the standard model. In Elette Boyle and Mohammad Mahmoody, editors, *TCC 2024, Part I*, volume 15364 of *LNCS*, pages 444–475. Springer, Cham, December 2024.

[36] Yevgeniy Dodis, Daniel Jost, and Antonio Marcedone. Compact key storage - A modern approach to key backup and delegation. In Leonid Reyzin and Douglas Stebila, editors, *CRYPTO 2024, Part II*, volume 14921 of *LNCS*, pages 75–109. Springer, Cham, August 2024.

[37] 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.

[38] 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, Cham, August 2017.

[39] Kevin Fu, Seny Kamara, and Yoshi Kohno. Key regression: Enabling efficient key distribution for secure distributed storage. In *NDSS 2006*. The Internet Society, February 2006.

[40] Roxana Geambasu, Tadayoshi Kohno, Amit A. Levy, and Henry M. Levy. Vanish: Increasing data privacy with self-destructing data. In Fabian Monrose, editor, *USENIX Security 2009*, pages 299–316. USENIX Association, August 2009.

[41] Eu-Jin Goh, Hovav Shacham, Nagendra Modadugu, and Dan Boneh. SiRiUS: Securing remote untrusted storage. In *NDSS 2003*. The Internet Society, February 2003.

[42] Oded Goldreich. Towards a theory of software protection and simulation by oblivious RAMs. In Alfred Aho, editor, *19th ACM STOC*, pages 182–194. ACM Press, May 1987.

[43] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions (extended abstract). In *25th FOCS*, pages 464–479. IEEE Computer Society Press, October 1984.

[44] Oded Goldreich, Shafi Goldwasser, and Silvio Micali. How to construct random functions. *Journal of the ACM*, 33(4):792–807, October 1986.

[45] Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. A concrete treatment of efficient continuous group key agreement via multi-recipient PKEs. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 1441–1462. ACM Press, November 2021.

[46] Nadia Heninger and Keegan Ryan. The hidden number problem with small unknown multipliers: Cryptanalyzing MEGA in six queries and other applications. In Alexandra Boldyreva and Vladimir Kolesnikov, editors, *PKC 2023, Part I*, volume 13940 of *LNCS*, pages 147–176. Springer, Cham, May 2023.

[47] Jonas Hofmann and Kien Tuong Truong. End-to-end encrypted cloud storage in the wild: A broken ecosystem. In Bo Luo, Xiaojing Liao, Jun Xu, Engin Kirda, and David Lie, editors, *ACM CCS 2024*, pages 3988–4001. ACM Press, October 2024.

[48] Mahesh Kallahalla, Erik Riedel, Ram Swaminathan, Qian Wang, and Kevin Fu. Plutus: Scalable secure file sharing on untrusted storage. In *Proceedings of the 2nd USENIX Conference on File and Storage Technologies*, FAST '03, page 29–42, USA, 2003. USENIX Association.

[49] 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.

[50] Karen Klein, Guillermo Pascual-Perez, Michael Walter, Chethan Kamath, Margarita Capretto, Miguel Cueto, Ilia Markov, Michelle Yeo, Joël Alwen, and Krzysztof Pietrzak. Keep the dirt: Tainted TreeKEM, adaptively and actively secure continuous group key agreement. In *2021 IEEE Symposium on Security and Privacy*, pages 268–284. IEEE Computer Society Press, May 2021.

[51] 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, Cham, May 2019.

[52] 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, Cham, April / May 2018.

[53] Jinyuan Li, Maxwell Krohn, David Mazières, and Dennis Shasha. Secure untrusted data repository (SUNDR). In *6th Symposium on Operating Systems Design & Implementation (OSDI 04)*, San Francisco, CA, December 2004. USENIX Association.

[54] Giorgia Azzurra Marson and Bertram Poettering. Practical secure logging: Seekable sequential key generators. In Jason Crampton, Sushil Jajodia, and Keith Mayes, editors, *ESORICS 2013*, volume 8134 of *LNCS*, pages 111–128. Springer, Berlin, Heidelberg, September 2013.

[55] Giorgia Azzurra Marson and Bertram Poettering. Even more practical secure logging: Tree-based seekable sequential key generators. In Miroslaw Kutylowski and Jaideep Vaidya, editors, *ESORICS 2014, Part II*, volume 8713 of *LNCS*, pages 37–54. Springer, Cham, September 2014.

[56] Mega Limited. Mega security white paper – third edition. https://mega.nz/SecurityWhitepaper.pdf, Jun 2022.

[57] Armin Namavari, Barry Wang, Sanketh Menda, Ben Nassi, Nirvan Tyagi, James Grimmelmann, Amy X. Zhang, and Thomas Ristenpart. Private hierarchical governance for encrypted messaging. In *2024 IEEE Symposium on Security and Privacy*, pages 2610–2629. IEEE Computer Society Press, May 2024.

[58] Bertram Poettering, Paul Rösler, Jörg Schwenk, and Douglas Stebila. SoK: Game-based security models for group key exchange. In Kenneth G. Paterson, editor, *CT-RSA 2021*, volume 12704 of *LNCS*, pages 148–176. Springer, Cham, May 2021.

[59] Raluca Ada Popa, Emily Stark, Steven Valdez, Jonas Helfer, Nickolai Zeldovich, and Hari Balakrishnan. Building web applications on top of encrypted data using mylar. In *11th USENIX Symposium on Networked Systems Design and Implementation (NSDI 14)*, pages 157–172, Seattle, WA, April 2014. USENIX Association.

[60] E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.3. RFC 8446 (Proposed Standard), August 2018.

[61] Phillip Rogaway. Authenticated-encryption with associated-data. In Vijayalakshmi Atluri, editor, *ACM CCS 2002*, pages 98–107. ACM Press, November 2002.

[62] Matteo Scarlata, Matilda Backendal, and Miro Haller. MFKDF: Multiple factors knocked down flat. In Davide Balzarotti and Wenyuan Xu, editors, *USENIX Security 2024*. USENIX Association, August 2024.

[63] Hossein Shafagh, Lukas Burkhalter, Sylvia Ratnasamy, and Anwar Hithnawi. Droplet: Decentralized authorization and access control for encrypted data streams. In Srdjan Capkun and Franziska Roesner, editors, *USENIX Security 2020*, pages 2469–2486. USENIX Association, August 2020.

[64] Nirvan Tyagi, Muhammad Haris Mughees, Thomas Ristenpart, and Ian Miers. BurnBox: Self-revocable encryption in a world of compelled access. In William Enck and Adrienne Porter Felt, editors, *USENIX Security 2018*, pages 445–461. USENIX Association, August 2018.

[65] Frank Wang, James Mickens, Nickolai Zeldovich, and Vinod Vaikuntanathan. Sieve: Cryptographically enforced access control for user data in untrusted clouds. In *13th USENIX Symposium on Networked Systems Design and Implementation (NSDI 16)*, pages 611–626, Santa Clara, CA, March 2016. USENIX Association.

A Extended Explanation of the Data Protection Mechanisms Table

This section offers an extended explanation of Table 1. The table offers a systematization of data protection mechanisms for group applications, comparing data in transit to data at rest.

The columns specify data actions. For data in transit, the actions describe the event where data *is sent* or *is received* by a member of the group. In other words, data transfer is not instantaneous but rather split into the action of sending data, and the action of receiving data (after some time, during which a compromise could happen). This split is useful for two reasons: (1) it captures data in transit, which is after sending but before receiving and (2) it allows us to model explicitly that key material can be discarded or rotated as soon as the message was received.[18] For data at rest, we consider the data actions where new data *is created or updated*, which implies that it is accessible to anyone currently in the group, or when existing data *is deleted*.

The rows of Table 1 specify the following five cases, which cover all possible time-points of compromise relative to the data actions.

- *Malicious former member*: an adversary-controlled user (who was either malicious from the start or compromised) was removed before the data action takes place.
- *Previously compromised member*: a group member that is part of the group during the data action and who recovered from a previous compromise (e.g., through key rotation) before the action.
- *Compromised active member*: the adversary is part of the group during the data action.
- *Not-yet-compromised member*: a group member that is part of the group during the data action and that will be compromised at some later point.
- *Malicious future member*: an adversary-controlled user will be added to the group after the data action (this can either be a user that is malicious from the start or an honest user added after the action and who is later compromised).

In order to uniquely identify scenarios and avoid redundancy, we make two assumptions.

First, we assume there is only one compromise. The table easily generalizes to a system with multiple compromises, where the union of all cells defines the set of necessary protection mechanisms. This also implies that if any of the compromises results in no available protection mechanism, then the data has to be considered compromised. Intuitively, this captures that the adversary only needs one successful attack avenue.

Second, we assume that no action relevant for data security happens between the compromise and the data action that are specified by the row and column respectively. For instance, the cell at the intersection of the column for the data action where data at rest *is deleted* and the row for a *previously compromised member* lists the protection mechanism for data that is removed while one of the current group members recovered from a compromise before the action. As we assume no other security relevant action happens between the "recovery from compromise" and the data deletion, this data is disclosed to the adversary because it was uploaded while or before the member was compromised. Otherwise, if the data would be created after the recovery (and before it is deleted), we would be in the scenario for the row *previously compromised member* and the column *is created or updated*, which is already captured by a different cell of the table.

Three examples of scenarios that were not explored by prior work on data at rest are the following:

[18] This is the analogy for data in transit to deletion of persistent data.

1. Column *is created or updated* and row *malicious former member* shows that persistent data that is created after the adversary was removed can be kept inaccessible to the adversary by rotating the group key material at or after the removal, before the new data is encrypted.

2. Column *is created or updated* and row *previously compromised member* shows that it is possible to achieve PCS for data that is created after a member recovered from a previous compromise by both ratcheting asymmetric key material and rotating symmetric keys.

3. Row *malicious future member* shows that one can protect any data from adversaries joining the group in the future by ratcheting the key material (e.g., in a hash chain) after it was used to encrypt the data.

B Standard Building Blocks

B.1 PRFs

A function family $\mathsf{F} : \{0,1\}^{xl} \times \{0,1\}^{yl} \to \{0,1\}^{zl}$ is a function with two arguments, usually called key and message. For each function family F, the *swapped* function $\bar{\mathsf{F}} : \{0,1\}^{yl} \times \{0,1\}^{xl} \to \{0,1\}^{zl}$ is defined as $\bar{\mathsf{F}}(y,x) = \mathsf{F}(x,y)$. That is, $\bar{\mathsf{F}}$ swaps the key for the message, and vice versa. We say that F is a secure pseudorandom function (PRF) if, for any uniformly random $x \in \{0,1\}^{xl}$, $\mathsf{F}(x,\cdot)$ is indistinguishable from a random function $f : \{0,1\}^{yl} \to \{0,1\}^{zl}$ for a computational adversary that does not know x. Following [22], we call F a *swap-PRF* if $\bar{\mathsf{F}}$ is a secure PRF.

Definition B.1 (PRF and swap-PRF security)**.** Let F be a function family. We define the advantage of an adversary $\mathcal{A}$ against the PRF security of F as

$$\mathbf{Adv}^{\text{prf}}_{\mathsf{F}}(\mathcal{A}) = \Pr[\mathbf{G}^{\text{prf-1}}_{\mathsf{F}}(\mathcal{A})] - \Pr[\mathbf{G}^{\text{prf-0}}_{\mathsf{F}}(\mathcal{A})] \ .$$

where game $\mathbf{G}^{\text{prf-}b}_{\mathsf{F}}$ is defined in Figure 15. Following [22], we similarly define the advantage against the swapped PRF security of F as

$$\mathbf{Adv}^{\text{swap-prf}}_{\mathsf{F}}(\mathcal{A}) = \Pr[\mathbf{G}^{\text{prf-1}}_{\bar{\mathsf{F}}}(\mathcal{A})] - \Pr[\mathbf{G}^{\text{prf-0}}_{\bar{\mathsf{F}}}(\mathcal{A})] \ .$$

Notice that the definitions are identical, except that for swapped PRF security the game is parameterized by the swapped function $\bar{\mathsf{F}}$. That is, swapped PRF security is equivalent to PRF security of the swapped function. Both games are given in the multi key setting [20], meaning that the adversary has access to a "$\mathcal{O}\texttt{New}$" oracle for generating new keys. The first input to the function oracle $\mathcal{O}\texttt{Fn}$ is an index which determines which out of these generated keys to use.

B.2 AEAD

We recall the syntax and security of an AEAD scheme from Rogaway [61], with the syntax modified to use implicit random nonces and security lifted to the multi-user setting [20].

Definition B.2 (Authenticated encryption with associated data)**.** An AEAD scheme $\mathsf{AEAD} = (\mathsf{Enc}, \mathsf{Dec})$ is a pair of algorithms. Associated to the scheme is a key length ℓ_a and a ciphertext space $\mathcal{C}$.

- Via $C \leftarrow\!\!\$\ \mathsf{Enc}(K, M, AD)$, the randomized algorithm encryption algorithm on input key $K \in \{0,1\}^{\ell_a}$, message $M \in \{0,1\}^*$, and associated data $AD \in \{0,1\}^*$, outputs a ciphertext $C \in \mathcal{C}$.

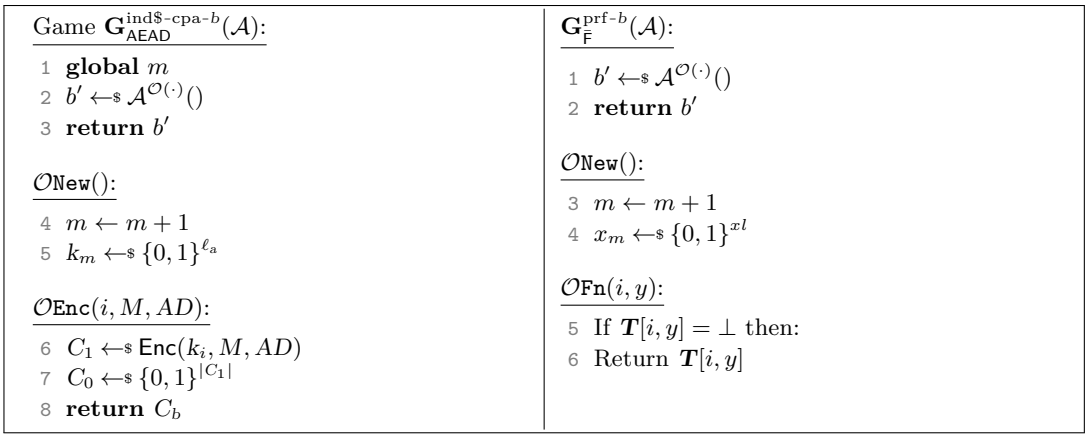

Figure 15: Game formalizing ind$-cpa security of an authenticated encryption scheme with associated data AEAD and PRF security for a function family $\mathsf{F} : \{0,1\}^{xl} \times \{0,1\}^{yl} \to \{0,1\}^{zl}$.

- Via $M/\perp \leftarrow \mathsf{Dec}(K, C, AD)$, the deterministic decryption algorithm on input a key $K \in \{0,1\}^{\ell_a}$, ciphertext $C \in \mathcal{C}$, and associated data $AD \in \{0,1\}^*$, outputs a message M or the error symbol $\perp$ if decryption fails.

Definition B.3 (ind$-cpa)**.** Let AEAD be an AEAD scheme and consider the game $\mathbf{G}^{\text{ind\$-cpa-}b}_{\mathsf{AEAD}}$ in Figure 15. The advantage of an adversary $\mathcal{A}$ against the *indistinguishability from random under chosen-ciphertext attack* of AEAD is defined as

$$\mathbf{Adv}^{\text{ind\$-cpa}}_{\mathsf{AEAD}}(\mathcal{A}) = \Pr[\mathbf{G}^{\text{ind\$-cpa-1}}_{\mathsf{AEAD}}(\mathcal{A})] - \Pr[\mathbf{G}^{\text{ind\$-cpa-0}}_{\mathsf{AEAD}}(\mathcal{A})] \ .$$

C Extended Comparison to Compact Key Storage

Dodis, Jost, and Marcedone (DJM) [36] propose two compact key storage (CKS) constructions—a *line scheme* and an *interval scheme*—to efficiently store a sequence of encryption secrets from a secure messaging application while preserving their forward security and post-compromise security. The encrypted messages can be taken directly from the messaging application and stored persistently on the backup server. The remaining problem and focus of CKS is to store the secrets used for encrypting messages with a compact (i.e., small) per-client storage, and a global, server-stored state that has linear size in the number of secrets. An explicit design goal of CKS is deduplication: instead of letting each group member individually back up secrets, CKS aims to have a single backup for all users in a group. This joint backup saves global storage space, while still preserving the access permissions from the secure messaging application. Access to the backed up secrets can be delegated to others by any user that has access to them.

CKS technical overview. Let $m_i, m_{i+1}, \ldots, m_j$ be the sequence of messages received by group members, encrypted with the corresponding secrets $s_i, s_{i+1}, \ldots, s_j$. The *line scheme* uses convergent encryption [37] to deterministically derive an encryption key K_j from the j-th messaging secret s_j and the previous encryption key K_{j-1}, and then uses K_j to encrypt s_j and K_{j-1} producing ciphertext c_j. This ciphertext is uploaded to the server. Once given access to a key K_j, a user can fetch and decrypt the corresponding ciphertext c_j to learn secret s_j to decrypt message m_j. The user further learns K_{j-1} and can recover m_{j-1} similarly as above, ultimately recovering all previous messages. (Note that the line scheme hence only supports all-or-nothing access to the full history of messages

and secrets.) Decrypting a specific message or checking that delegated keys are consistent with locally stored ones takes linear time.

The *interval scheme* uses a binary tree with the messaging secrets in the leaves. Similar to the line scheme, convergent encryption is used to derive parent nodes, where the ciphertext associated with a node is the encryption of the keys necessary to decrypt the ciphertexts of the two child nodes. Hence intervals of secrets can be communicated by sharing the keys for the minimal set of parent nodes necessary to derive all keys in the interval. This reduces the running time to only be logarithmic in the number of secrets but comes at the cost of increasing the communication complexity from being constant to logarithmic. Moreover, access can be confined to intervals instead of always granting access to all previous messages.

Threat model. CKS considers a malicious server and malicious insiders for their adversarial modeling. The security of their constructions, however, depends on the security of the secure messaging scheme for which they back up the messages. If CKS is instantiated with a secure messaging protocol that is not fully secure against active adversaries, such as MLS [18], the security of CKS is also weakened. In fact, the construction of practical group messaging schemes with fully active security is an open problem.

In contrast, in our work we consider a standalone security game that does not rely on the security of an external protocol, but only capture passive adversaries. As we discuss in Section 5.2, one reason for this choice is that building authenticated GKP from standard GKP is straightforward. Nevertheless, as long as Grappa is instantiated using CGKA, it cannot achieve fully active security efficiently either.

Setup assumptions and costs. Our constructions achieve IAC to protect past keys from new members through explicit key rotation, but not FS. This ensures that DKR-based interval scheme $\mathsf{D}[\mathsf{H}, \mathsf{F}]$ has compact local storage. In order to achieve FS, we would need to pick fresh keys in every epoch, which comes at linear storage cost. CKS achieves to keep their local storage compact by requiring a global server state that grows linearly with the number of backed up keys. Our solution does not mandate any global server state and can therefore be used in applications without a central server, as long as a delivery service can be instantiated in a decentralized manner.

The interval scheme of CKS is limited to a predefined constant number of epochs, while our primitives support key rotations that enable an infinite number of epochs. CKS supports receiving secrets out-of-order, storing them locally before embedding them in the compact storage. GKP forces group admins to have processed the most recent updates before they are able to release new keys. Implicitly, this also applies to CGKA-based secure messaging applications that produce the secrets for CKS. Our generalized syntax for interval schemes only allows expanding intervals with consecutive (in-order) updates, though one could consider an extension to locally cache future key interval updates until intermediate values are received.

GKP for secure messaging backups. GKP can be used instead of CKS to encrypt and outsource the storage of messaging secrets. As in CKS, we consider the exchange of messages $m_1, m_2, \ldots$ between all parties of a chat group. The messages are encrypted with the secrets $s_1, s_2, \ldots$ resulting in ciphertexts $\tilde{c}_1, \tilde{c}_2, \ldots$ that are outsourced to a cloud storage provider. The GKP scheme is synchronized with all changes in the secure messaging application: when participants are added/removed or keys are rotated, the same operation is executed for GKP. To encrypt message m_i, the application requests epoch key k_i from GKP and compute $\tilde{c}_i \leftarrow \mathsf{Enc}(k_i, s_i)$.

As CKS is tailored to this application, it is unsurprising that GKP is less efficient as it does not leverage the existing FS and PCS properties of the encrypted secrets. However, it is possible to use the GKP scheme to maintain the FS and PCS of secrets by rotating chains regularly, or to use our relaxations to achieve more efficient solutions that provide IAC for the encrypted messages.

In contrast to using GKP for secure message backups, CKS cannot be used to instantiate interval schemes or GKP, nor achieve IAC for general persistent data, for the following reasons. GKP is expected to produce indistinguishable keys, without making any assumptions about the amount of entropy in the data to be encrypted. Due to their goal of deduplication, CKS use convergent encryption to deterministically derive keys from messaging secrets and previous keys. Clearly, this makes the derived keys dependent on the encrypted secrets. Consequently, they do not achieve key indistinguishability, and need to further assume that the encrypted data has high entropy. This is the case for messaging secrets, but not generally true for files of an E2EE shared folder and other persistent data. Hence, we conclude that CKS can only preserve properties of the encrypted data, but since they do not generate independent keys they cannot provide additional IAC guarantees to existing persistent data without any inherent guarantees. Additionally, when using CKS as an interval scheme the linear server storage of encrypted secrets would need to be stored locally, which undoes all benefits from a compact client-storage and has no better storage complexity than storing all keys (scheme $\mathsf{trivial}$ from Section 4.6) Moreover, the CKS syntax lacks the group membership operations required for GKP.

Later, Dodis and Jost improved on CKS in follow-up work [35], where they replaced convergent encryption with a new primitive called "trapdoor KDF". Their main contribution is to prove this version of CKS secure in the standard model, but the keys are still derived from the encrypted data (the messaging secrets) and the previous trapdoor KDF state, and they still require linear server storage for ciphertexts. Hence, the above discussion of limitations for using their construction for the GKP use case still applies.

D Double-PRF Security: Proof of Lemma 4.5

We now provide the proof of Lemma 4.5. Recall that the Lemma states that a function family $\mathsf{F} : \{0,1\}^{xl} \times \{0,1\}^{yl} \to \{0,1\}^{zl}$ is a double-PRF if and only if it is a PRF and a swap-PRF.

Proof of Lemma 4.5. We first note that if F is a double-PRF, then it is trivially a PRF. This is evident from the double-PRF game in Figure 5, since an adversary can simply ignore oracles $\mathcal{O}\mathtt{NewY}$ and $\mathcal{O}\mathtt{FnY}$ and only make queries to oracles $\mathcal{O}\mathtt{NewX}$ and $\mathcal{O}\mathtt{FnX}$, in which case the dual PRF game behaves exactly as the PRF game. By symmetry the same holds for the swapped PRF security of F. This shows the backwards implication.

For the forward direction, we claim that for any adversary $\mathcal{A}$ against the double-PRF security of F, there exists adversaries $\mathcal{B}_1$, $\mathcal{B}_2$ and $\mathcal{B}_3$ such that

$$\mathbf{Adv}^{\text{dbl-prf}}_{\mathsf{F}}(\mathcal{A}) \leq \mathbf{Adv}^{\text{prf}}_{\mathsf{F}}(\mathcal{B}_1) + \mathbf{Adv}^{\text{swap-prf}}_{\mathsf{F}}(\mathcal{B}_2) + \frac{\mathcal{Q}_{\mathcal{O}\mathtt{FnX}}(\mathcal{A}) \cdot \mathcal{Q}_{\mathcal{O}\mathtt{NewY}}(\mathcal{A})}{2^{zl}} + \mathbf{Adv}^{\text{swap-prf}}_{\mathsf{F}}(\mathcal{B}_3). \quad (1)$$

W.l.o.g. we restrict our attention to adversaries which do not repeat queries to oracles $\mathcal{O}\mathtt{FnX}$, $\mathcal{O}\mathtt{FnY}$ and $\mathcal{O}\mathtt{FnXY}$, to simplify notation. The proof proceeds by a series of game hops. We let game G_0 and G_3 be identical to $\mathbf{G}^{\text{dbl-prf-1}}_{\mathsf{F}}$ and $\mathbf{G}^{\text{dbl-prf-0}}_{\mathsf{F}}$, respectively.

Next, we define game G_1 and G_2 in Figure 16. Game G_2 is identical to G_0, except that oracles $\mathcal{O}\mathtt{FnX}$ and $\mathcal{O}\mathtt{FnXY}$ return random strings in $\{0,1\}^{zl}$ instead of PRF evaluations. Additionally, a

Game $\boxed{\mathrm{G}_1}$, G_2:

$b' \leftarrow\!\!\$\ \mathcal{A}^{\mathcal{O}(\cdot)}()$
return b'

$\mathcal{O}\texttt{NewX}()$:

$n \leftarrow n+1$
$x_n \leftarrow\!\!\$\ \{0,1\}^{xl}$

$\mathcal{O}\texttt{NewY}()$:

$m \leftarrow m+1$
$y_m \leftarrow\!\!\$\ \{0,1\}^{yl}$
$\mathcal{S} \xleftarrow{\cup} \{y_m\}$

$\mathcal{O}\texttt{FnX}(i,y)$:

$\boldsymbol{T}_x[i,y] \leftarrow\!\!\$\ \{0,1\}^{zl}$
If $\exists j \le m : y = y_j$ and $\boldsymbol{T}_{xy}[i,j] \neq \perp$: // First condition true if $y \in \mathcal{S}$
  $\mathsf{bad} \leftarrow 1$; $\boxed{\boldsymbol{T}_x[i,y] \leftarrow \boldsymbol{T}_{xy}[i,j]}$
$\mathcal{Q} \xleftarrow{\cup} \{y\}$
Return $\boldsymbol{T}_x[i,y]$

$\mathcal{O}\texttt{FnY}(x,i)$:

$\boldsymbol{T}_y[x,i] \leftarrow \mathsf{F}(x, y_i)$
Return $\boldsymbol{T}_y[x,i]$

$\mathcal{O}\texttt{FnXY}(i,j)$:

$\boldsymbol{T}_{xy}[i,j] \leftarrow\!\!\$\ \{0,1\}^{zl}$
If $y_j \in \mathcal{Q}$ and $\boldsymbol{T}_x[i,y_j] \neq \perp$ then:
  $\mathsf{bad} \leftarrow 1$; $\boxed{\boldsymbol{T}_{xy}[i,j] \leftarrow \boldsymbol{T}_x[i,y_j]}$
Return $\boldsymbol{T}_{xy}[i,j]$

Figure 16: Games G_1 and G_2 for the proof of Lemma 4.5.

Game G_2':

$\mathcal{A}^{\mathcal{O}(\cdot)}()$
Return $\mathcal{Q} \cap \mathcal{S} \neq \emptyset$

$\mathcal{O}\texttt{NewX}()$:

$n \leftarrow n+1$
$x_n \leftarrow\!\!\$\ \{0,1\}^{xl}$

$\mathcal{O}\texttt{NewY}()$:

$m \leftarrow m+1$
$y_m \leftarrow\!\!\$\ \{0,1\}^{yl}$
$\mathcal{S} \xleftarrow{\cup} \{y_m\}$

$\mathcal{O}\texttt{FnX}(i,y)$:

$\boldsymbol{T}_x[i,y] \leftarrow\!\!\$\ \{0,1\}^{zl}$
$\mathcal{Q} \xleftarrow{\cup} \{y\}$
Return $\boldsymbol{T}_x[i,y]$

$\mathcal{O}\texttt{FnY}(x,i)$:

$\boldsymbol{T}_y[x,i] \leftarrow \mathsf{F}(x, y_i)$
Return $\boldsymbol{T}_y[x,i]$

$\mathcal{O}\texttt{FnXY}(i,j)$:

$\boldsymbol{T}_{xy}[i,j] \leftarrow\!\!\$\ \{0,1\}^{zl}$
Return $\boldsymbol{T}_{xy}[i,j]$

Figure 17: Game G_2' for the proof of Lemma 4.5.

boolean flag bad is set to 1 if adversary $\mathcal{A}$ makes queries $\mathcal{O}\texttt{FnX}(i,y)$ and $\mathcal{O}\texttt{FnXY}(i,j)$ such that $y = y_j$ for a y_j generated by oracle $\mathcal{O}\texttt{NewY}$. That is, bad is set to 1 if $\mathcal{A}$ "guesses" a key y_j for the swapped function and uses y_j as the input in a query to the PRF oracle $\mathcal{O}\texttt{FnX}$. Game G_1 is identical to G_2, except that if bad is set to 1, the current query is responded to with a value which is consistent with the prior query which used the same key. That is, instead of returning independent, uniformly random points in $\{0,1\}^{zl}$, queries $\mathcal{O}\texttt{FnX}(i,y)$ and $\mathcal{O}\texttt{FnXY}(i,j)$ for which $y = y_j$ give the same response. Additionally, we define game G_2' in Figure 17. G_2' is identical to game G_2, except that it does not return the bit guess of the adversary, but instead returns 1 if a query $\mathcal{O}\texttt{FnX}(i,y)$ is made such that y is one of the keys generated by oracle $\mathcal{O}\texttt{NewY}$.

We proceed to prove Equation (1), as follows. By definition, $\mathbf{Adv}^{\text{dbl-prf}}_{\mathsf{F}}(\mathcal{A}) = |\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_3]|$. Hence, through standard equation rewriting, $\mathbf{Adv}^{\text{dbl-prf}}_{\mathsf{F}}(\mathcal{A}) \le |\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_1]| + |\Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_2]| + |\Pr[\mathrm{G}_2] - \Pr[\mathrm{G}_3]|$. Games G_1 and G_2 are identical-until-bad, so we have $|\Pr[\mathrm{G}_1] - \Pr[\mathrm{G}_2]| \le \Pr[\mathrm{G}_2 \text{ sets } \mathsf{bad}]$ by the fundamental lemma of game playing [23]. We now construct adversaries $\mathcal{B}_1$,

$\mathcal{B}_2$ and $\mathcal{B}_3$ such that

$$|\Pr[\mathrm{G}_0] - \Pr[\mathrm{G}_1]| \leq \mathbf{Adv}^{\mathrm{prf}}_{\mathsf{F}}(\mathcal{B}_1) \tag{2}$$

$$\Pr[\mathrm{G}_2 \text{ sets } \mathsf{bad}] \leq \mathbf{Adv}^{\text{swap-prf}}_{\mathsf{F}}(\mathcal{B}_2) + \frac{\mathcal{Q}_{\mathcal{O}\mathtt{FnX}}(\mathcal{A}) \cdot \mathcal{Q}_{\mathcal{O}\mathtt{NewY}}(\mathcal{A})}{2^{zl}} \tag{3}$$

$$|\Pr[\mathrm{G}_2] - \Pr[\mathrm{G}_3]| \leq \mathbf{Adv}^{\text{swap-prf}}_{\mathsf{F}}(\mathcal{B}_3). \tag{4}$$

Together, this proves (1).

Adversary $\mathcal{B}_1$ runs $\mathcal{A}$, acting as the challenger in game G_0, except that it relays queries to oracles $\mathcal{O}\mathtt{NewX}$ to oracle $\mathcal{O}\mathtt{New}$ and queries to oracles $\mathcal{O}\mathtt{FnX}$ and $\mathcal{O}\mathtt{FnXY}$ to oracle $\mathcal{O}\mathtt{Fn}$ in the PRF game. For the latter, it replaces the j in a query $\mathcal{O}\mathtt{FnXY}(i,j)$ by y_j which it samples itself when simulating oracle $\mathcal{O}\mathtt{NewY}$. When adversary $\mathcal{A}$ halts and returns b^*, $\mathcal{B}_1$ also halts and returns b^*. This way $\mathcal{B}_1$ simulates game G_0 when the hidden bit b in the PRF game played by $\mathcal{B}$ is 1 and G_1 when $b = 0$. Additionally, $\Pr\left[\mathbf{G}^{\text{prf-1}}_{\mathsf{F}}(\mathcal{B}_1)\right] = \Pr[\mathrm{G}_0]$ and $\Pr\left[\mathbf{G}^{\text{prf-0}}_{\mathsf{F}}(\mathcal{B}_1)\right] = \Pr[\mathrm{G}_1]$, giving Equation (2).

We proceed to show Equation (3). First, we claim that $\Pr[\mathrm{G}_2 \text{ sets } \mathsf{bad}] \leq \Pr[\mathrm{G}'_2(\mathcal{A})]$. To see this, note that the event "G_2 sets bad" implies that $\mathcal{Q} \cap \mathcal{S} \neq \emptyset$ at the end of game G_2, where set $\mathcal{Q}$ contains the y-values queried to oracle $\mathcal{O}\mathtt{FnX}$ and set $\mathcal{S}$ contains the y-values generated by queries to oracle $\mathcal{O}\mathtt{NewY}$. Since games G_2 and G'_2 are identical except for the return statement (in particular, sets $\mathcal{Q}$ and $\mathcal{S}$ are defined identically in both games), and $\Pr[\mathrm{G}'_2(\mathcal{A})] = \Pr[\mathcal{Q} \cap \mathcal{S} \neq \emptyset \text{ at the end of game } \mathrm{G}'_2]$ by definition, this shows the claimed inequality. Next, we construct adversary $\mathcal{B}_2$ against the swap-prf security of F such that

$$\Pr\left[\mathbf{G}^{\text{prf-1}}_{\bar{\mathsf{F}}}(\mathcal{B}_2)\right] \geq \Pr[\mathrm{G}'_2(\mathcal{A})] \text{ and} \tag{5}$$

$$\Pr\left[\mathbf{G}^{\text{prf-0}}_{\bar{\mathsf{F}}}(\mathcal{B}_2)\right] \leq \frac{\mathcal{Q}_{\mathcal{O}\mathtt{FnX}}(\mathcal{A}) \cdot \mathcal{Q}_{\mathcal{O}\mathtt{NewY}}(\mathcal{A})}{2^{zl}}, \tag{6}$$

which together implies Equation (3).

Adversary $\mathcal{B}_2$, shown in Figure 18, works as follows. It runs $\mathcal{A}$, acting as the challenger in game G'_2, except that it forwards queries to oracle $\mathcal{O}\mathtt{NewY}$ to its $\mathcal{O}\mathtt{New}$ and responds to queries to oracle $\mathcal{O}\mathtt{FnY}$ by forwarding them (with the inputs swapped, due to the definition of swap-prf as the prf security of $\bar{\mathsf{F}}$) to oracle $\mathcal{O}\mathtt{Fn}$. Additionally, adversary $\mathcal{B}_2$ keeps a counter m which counts the number of $\mathcal{O}\mathtt{NewY}$ queries made by $\mathcal{A}$, and sets $\mathcal{Q}_x$ and $\mathcal{Q}_y$ which store the x- and y-values used in queries by adversary $\mathcal{A}$ to oracles $\mathcal{O}\mathtt{FnY}$ and $\mathcal{O}\mathtt{FnX}$, respectively. When adversary $\mathcal{A}$ halts, adversary $\mathcal{B}_2$ uses the y-values in $\mathcal{Q}_y$ to check the consistency of its $\mathcal{O}\mathtt{Fn}$-oracle with the function F. To do this, it chooses an arbitrary $x \in \{0,1\}^{xl} \setminus \mathcal{Q}_x$. Then, for each key index i in $[m]$, $\mathcal{B}_2$ queries oracle $\mathcal{O}\mathtt{Fn}(i,x)$ and assigns the output to z_i. Adversary $\mathcal{B}_2$ returns 1 if for any $i \in [m]$ and any $y \in \mathcal{Q}_y$, $z_i = \mathsf{F}(x,y)$.

This way, when adversary $\mathcal{B}_2$ is playing game $\mathbf{G}^{\text{prf-1}}_{\bar{\mathsf{F}}}$, it perfectly simulates game G'_2 for adversary $\mathcal{A}$. Additionally, $\Pr\left[\mathbf{G}^{\text{prf-1}}_{\bar{\mathsf{F}}}(\mathcal{B}_2)\right] \geq \Pr[\mathrm{G}'_2(\mathcal{A})]$, since if $\mathcal{A}$ has queried oracle $\mathcal{O}\mathtt{FnX}$ on a $y^* \in \{0,1\}^{yl}$ such that $y^* \in \mathcal{S}$ (meaning that $\mathcal{A}$ has effectively "guessed" one of the y-values generated by oracle $\mathcal{O}\mathtt{NewY}$ and that $y^* \in \mathcal{Q} \cap \mathcal{S} \neq \emptyset$), then there exists an $i \in [m]$ s.t. $y^* = y_i$ and for this i, $z_i = \mathcal{O}\mathtt{Fn}(i,x) = \mathsf{F}(x,y_i)$. Hence, when $\mathcal{B}_2$ loops through $\mathcal{Q}_y$ it will return 1 when it reaches y^*. If instead adversary $\mathcal{B}_2$ is playing game $\mathbf{G}^{\text{prf-0}}_{\bar{\mathsf{F}}}$, then $\mathcal{B}_2$ will return 1 with probability at most $\mathcal{Q}_{\mathcal{O}\mathtt{FnX}}(\mathcal{A}) \cdot \mathcal{Q}_{\mathcal{O}\mathtt{NewY}}(\mathcal{A})/2^{zl}$. To see this, we note $m = \mathcal{Q}_{\mathcal{O}\mathtt{NewY}}(\mathcal{A})$ and that $|\mathcal{Q}_y| \leq \mathcal{Q}_{\mathcal{O}\mathtt{FnX}}(\mathcal{A})$. We have that for each z_i defined as above and all $y \in \mathcal{Q}_y$, $\Pr[z_i = \mathsf{F}(x,y)] = 1/2^{zl}$ since x has not been queried to oracle $\mathcal{O}\mathtt{Fn}$, meaning that each z_i is a new independent random variable in $\{0,1\}^{zl}$. Let $\mathcal{F}(x, \mathcal{Q}_y) := \{\mathsf{F}(x,y) \text{ st. } y \in \mathcal{Q}_y\}$. Then for each $i \in [m]$, $\Pr[z_i \in \mathcal{F}(x, \mathcal{Q}_y)] \leq \mathcal{Q}_{\mathcal{O}\mathtt{FnX}}(\mathcal{A})/2^{zl}$ by the union bound. Hence $\Pr\left[\mathbf{G}^{\text{prf-0}}_{\bar{\mathsf{F}}}(\mathcal{B}_2)\right] = \Pr[\exists i \in [m] \text{ st. } z_i \in \mathcal{F}(x, \mathcal{Q}_y)] \leq \mathcal{Q}_{\mathcal{O}\mathtt{NewY}}(\mathcal{A}) \cdot \mathcal{Q}_{\mathcal{O}\mathtt{FnX}}(\mathcal{A})/2^{zl}$, again by the union bound. This proves the claimed bounds in Equations (5) and (6).

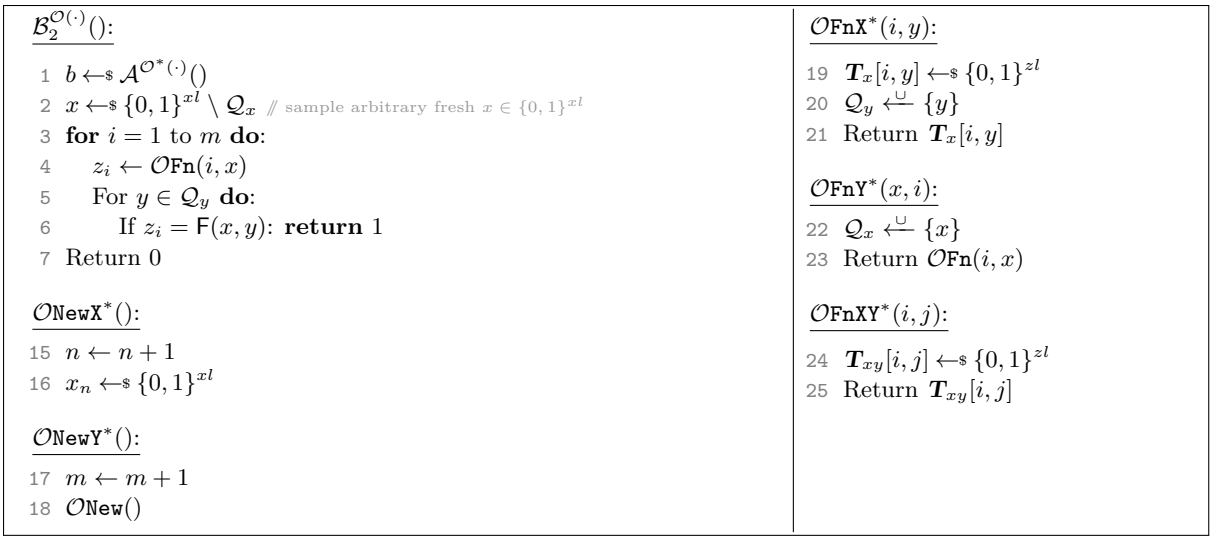

Figure 18: Adversary $\mathcal{B}_2$ for the proof of Lemma 4.5.

Finally, we construct adversary $\mathcal{B}_3$ against the swap-prf security of F to prove Equation (4). Adversary $\mathcal{B}_3$ functions analogously to adversary $\mathcal{B}_1$; it runs $\mathcal{A}$, acting as the challenger in game G_2, but relays $\mathcal{O}\texttt{NewY}$- and $\mathcal{O}\texttt{FnY}$-queries to oracles $\mathcal{O}\texttt{New}$ and $\mathcal{O}\texttt{Fn}$, respectively. When adversary $\mathcal{A}$ halts and returns b^*, $\mathcal{B}_3$ also halts and outputs b^*. This way $\mathcal{B}_3$ simulates game G_2 for $\mathcal{A}$ when it plays game $\mathbf{G}_{\bar{\mathsf{F}}}^{\text{prf-1}}$, and game G_3 when it plays game $\mathbf{G}_{\bar{\mathsf{F}}}^{\text{prf-0}}$. Hence, we have $\Pr\left[\mathbf{G}_{\bar{\mathsf{F}}}^{\text{prf-1}}(\mathcal{B}_3)\right] = \Pr[\mathrm{G}_2]$ and $\Pr\left[\mathbf{G}_{\bar{\mathsf{F}}}^{\text{prf-0}}(\mathcal{B}_3)\right] = \Pr[\mathrm{G}_3]$. This concludes the proof. □

E Key Indistinguishability of $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$: Proof of Theorem 4.6

In the following, OWC and F are a one-way chain and a function family, respectively, and $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ is the DKR-based interval scheme defined in Figure 6. Adversary $\mathcal{A}$ is an adversary against the key indistinguishability of $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$, as per the theorem statement.

Proof of Theorem 4.6. The proof proceeds through a sequence of games G_0–G_3, shown in Figure 19. The games only differ from game $\mathbf{G}_{\mathsf{D}[\mathsf{OWC},\mathsf{F}]}^{\text{iac}}$ and each other in the main procedure, so for simplicity, we only display these. Throughout, the oracles remain the same as in game $\mathbf{G}_{\mathsf{D}[\mathsf{OWC},\mathsf{F}]}^{\text{iac}}$, shown in Figure 4, except for $\mathcal{O}\texttt{ExposeAux}$ which is changed in G_1.

We begin with game G_0, which is identical to $\mathbf{G}_{\mathsf{INT},\mathsf{Safe}}^{\text{iac-1}}$ (the "real" game), except that we expand the procedure GetKey to show the details of the instantiation $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ (Figure 6). Naturally, this does not logically change the game. Therefore, we have that

$$\Pr[\mathbf{G}_{\mathsf{INT},\mathsf{Safe}}^{\text{iac-1}}(\mathcal{A})] = \Pr[\mathrm{G}_0(\mathcal{A})]\ . \tag{7}$$

Early safety checks and modification of $\mathcal{O}\texttt{ExposeAux}$ in stage 2. Recall that the winning condition checks that the challenge epoch e^*, requested by adversary $\mathcal{A}$ in the first stage of the game, is not compromised by the simultaneously requested exposed intervals, or keys that the adversary obtained via oracle $\mathcal{O}\texttt{GetKey}$. Additionally, the $\mathsf{Safe}(e^*)$ predicate checks that there are sufficient blocks surrounding epoch e^* such that at least one of the forward and backward chain is safe, taking

$\underline{\mathrm{G}_0}$:

$(gst, aux) \leftarrow\!\$\ \mathsf{Init}()$
$\boldsymbol{B}[0] \leftarrow ||$
$(e^*, I) \leftarrow\!\$\ \mathcal{A}^{\mathcal{O}(\cdot)}()$
for $i \in I$ **do**:
  $(st_i, l, r) \leftarrow \boldsymbol{ST}[i]$; $\mathcal{S}_{int} \xleftarrow{\cup} \{st_i\}$
  for $e = l$ to r **do**: $\boldsymbol{X}_{st}[e] \leftarrow 1$
$gst.\mathcal{F} \leftarrow \mathsf{GetFChains}(gst.\mathcal{F}, 0, gst.e_{max})$
$gst.\mathcal{B} \leftarrow \mathsf{GetBChains}(st.\mathcal{B}, 0, st.e_{max})$
$[(\cdot, fs)] \leftarrow \mathsf{GetFChains}(st.\mathcal{F}, e^*, e^*)$
$[(\cdot, bs, \cdot)] \leftarrow \mathsf{GetBChains}(st.\mathcal{B}, e^*, e^*)$
$fk \leftarrow \mathsf{OWC.GetKey}(fs)$
$bk \leftarrow \mathsf{OWC.GetKey}(bs)$
$k_{e^*} \leftarrow \mathsf{F}(fk, bk)$
// Stage 2:
$b' \leftarrow\!\$\ \mathcal{A}^{\mathcal{O}(\cdot)}(k_b, \mathcal{S}_{int})$
return $\neg \boldsymbol{X}_k[e^*] \wedge b' \wedge \mathsf{Safe}(e^*)$

$\underline{\mathcal{O}\texttt{ExposeAux}(e)}$:

$\boldsymbol{X}_{aux}[e] \leftarrow 1$
return $\boldsymbol{A}[e]$

$\underline{\mathrm{G}_1,\ \mathrm{G}_2,\ \mathrm{G}_3}$:

$(gst, aux) \leftarrow\!\$\ \mathsf{Init}()$
$\boldsymbol{B}[0] \leftarrow ||$
$(e^*, I) \leftarrow\!\$\ \mathcal{A}^{\mathcal{O}(\cdot)}()$
for $i \in I$ **do**:
  $(st_i, l, r) \leftarrow \boldsymbol{ST}[i]$; $\mathcal{S}_{int} \xleftarrow{\cup} \{st_i\}$
  for $e = l$ to r **do**: $\boldsymbol{X}_{st}[e] \leftarrow 1$
if $\neg(\mathsf{SafeFC}(e^*) \vee \mathsf{SafeBCAux}(e^*))$
  $\vee \boldsymbol{X}_{st}[e^*]$: **return** 0 // $\mathrm{G}_1 - \mathrm{G}_3$
$gst.\mathcal{F} \leftarrow \mathsf{GetFChains}(gst.\mathcal{F}, 0, gst.e_{max})$
$gst.\mathcal{B} \leftarrow \mathsf{GetBChains}(st.\mathcal{B}, 0, st.e_{max})$
$[(\cdot, fs)] \leftarrow \mathsf{GetFChains}(st.\mathcal{F}, e^*, e^*)$
$[(\cdot, bs, \cdot)] \leftarrow \mathsf{GetBChains}(st.\mathcal{B}, e^*, e^*)$
$fk \leftarrow \mathsf{OWC.GetKey}(fs)$
$bk \leftarrow \mathsf{OWC.GetKey}(bs)$
if $\mathsf{SafeFC}(e^*)$: $fk \leftarrow\!\$\ \{0,1\}^{\ell_s}$ // $\mathrm{G}_2, \mathrm{G}_3$
if $\mathsf{SafeBCAux}(e^*)$: $bk \leftarrow\!\$\ \{0,1\}^{\ell_s}$ // $\mathrm{G}_2, \mathrm{G}_3$
$k_{e^*} \leftarrow \mathsf{F}(fk, bk)$ // $\mathrm{G}_1, \mathrm{G}_2$
$k_{e^*} \leftarrow\!\$\ \{0,1\}^{\ell}$ // G_3
// Stage 2:
$b' \leftarrow\!\$\ \mathcal{A}^{\mathcal{O}(\cdot)}(k_b, \mathcal{S}_{int})$
return $\neg \boldsymbol{X}_k[e^*] \wedge b'$

$\underline{\mathcal{O}\texttt{ExposeAux}(e)\ \text{(stage 1)}}$:

$\boldsymbol{X}_{aux}[e] \leftarrow 1$
return $\boldsymbol{A}[e]$

$\underline{\mathcal{O}\texttt{ExposeAux}(e)\ \text{(stage 2)}}$:

if $\forall e_l \leq e : (\boldsymbol{B}[e_l] \in \{\mapsto, ||\})$
  $\wedge (\exists e' \in [e_l, e^*], \boldsymbol{X}_{st}[e'] \neq 0)$:
  abort // $\mathrm{G}_1 - \mathrm{G}_3$
return $\boldsymbol{A}[e]$

$\mathsf{Safe}(e^*) = \mathsf{SafeFC}(e^*) \vee (\mathsf{SafeBC}(e^*) \wedge \mathsf{SafeAux}(e^*))$.
$\mathsf{SafeFC}(e^*) = \forall e_x < e^* : \boldsymbol{X}_{st}[e_x],\ \exists e_{fl} : (e_x < e_{fl} \leq e^*) \ \wedge\ (\boldsymbol{B}[e_{fl}] \in \{\mapsto, ||\})$.
$\mathsf{SafeBC}(e^*) = \forall e_x > e^* : \boldsymbol{X}_{st}[e_x],\ \exists e_{fl} : (e^* < e_{fl} \leq e_x) \ \wedge\ (\boldsymbol{B}[e_{fl}] \in \{\mapsto, ||\})$.
$\mathsf{SafeAux}(e^*) = \forall e_x : \boldsymbol{X}_{aux}[e_x], \exists e_{fl} : (\boldsymbol{B}[e_{fl}] \in \{\mapsto, ||\})$
  $\wedge \big((e^* < e_{fl} \leq e_x) \vee (e_x < e_{fl} \leq e^*)\big)$.
$\mathsf{SafeBCAux}(e^*) = \mathsf{SafeBC}(e^*) \ \wedge\ \mathsf{SafeAux}(e^*)$

Figure 19: Main procedures of games G_0–G_3 for the proof of Theorem 4.6. Commented lines are only present in the indicated game.

into account auxiliary states obtained via $\mathcal{O}\texttt{ExposeAux}$ (which in $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ translates to backward chain exposure). At the bottom of Figure 19, we split Safe into three logical parts.

Observe that at the end of the first stage, all the queries which determine whether the state of the challenge epoch is safe (except for the stage-2 $\mathcal{O}\texttt{ExposeAux}$ and $\mathcal{O}\texttt{GetKey}$ queries) have already been made. In particular, all exposed intervals I are fixed (hence also $\boldsymbol{X}_{st}$), as are all relevant chain rotations. (Chain rotations requested by the adversary in the second stage can only happen in new epochs, and hence do not help protect the challenge epoch from the states which are exposed in the first stage.).

In game G_1, we perform the safety predicate checks early (line 49 of Figure 19), leaving only the check that the key itself has not been exposed and the $\mathsf{SafeAux}$ check in line 61 of Figure 19 together with the trivial exposure condition $\neg\boldsymbol{X}_{st}[e^*]$.

Given the above observation, G_0 and G_1 are identical except for one detail: the queries made to $\mathcal{O}\texttt{ExposeAux}$ in the second stage of the game may modify $\mathsf{SafeAux}$ after it is checked. To address this issue, we also modify the $\mathcal{O}\texttt{ExposeAux}$ oracle *only in the second stage of the game*, such that it aborts the game (returning 0) if the exposure query leads to a violation of $\mathsf{SafeAux}$, instead of waiting for the adversary to finish the game. The oracle also no longer writes on $\boldsymbol{X}_{aux}$. With these changes, both the $\boldsymbol{X}_{aux}$ and the $\boldsymbol{X}_{st}$ are fixed at the end of the first stage of the game, meaning that we can safely check $\mathsf{SafeAux}$ at the end of stage 1 without changing the winning probability of the adversary.

To see that the games are still fully equivalent, note that any second stage query to $\mathcal{O}\texttt{ExposeAux}$ that ends up in a violation of $\mathsf{SafeAux}$ in G_0 will cause the game to abort in G_1, and viceversa. Therefore, we have

$$\Pr[\mathrm{G}_0(\mathcal{A})] = \Pr[\mathrm{G}_1(\mathcal{A})] \,. \tag{8}$$

Indistinguishability with forward security of OWC. Game G_2 is identical to G_1, except that it additionally includes lines 56 and 57, which overwrite the keys generated by the forward and/or backward chain at the challenge epoch e^* by uniformly random strings, *if* the corresponding chain is "safe", i.e., the state at epoch e^* has not been exposed to $\mathcal{A}$. Note that, by the early check on line 49, at least one of the two chains is guaranteed to be safe, unless adversary $\mathcal{A}$ violated the safety predicate Safe. In the following, we assume w.l.o.g. that $\mathcal{A}$ executes such that $\mathsf{Safe}(e^*)$ is true. (If it does not, it has advantage 0, so the claimed bound below is trivially true.)

We construct an adversary $\mathcal{B}_{\mathsf{OWC}}$ against the key indistinguishability with forward security of OWC such that

$$\Pr[\mathrm{G}_1(\mathcal{A})] - \Pr[\mathrm{G}_2(\mathcal{A})] \leq \mathbf{Adv}^{\text{ind-fs}}_{\mathsf{OWC},N}(\mathcal{B}_{\mathsf{OWC}}) \,. \tag{9}$$

Adversary $\mathcal{B}_{\mathsf{OWC}}$ acts as the challenger in game G_1, with the following differences. First, whenever the game runs $\mathsf{OWC.Gen}(N')$ (during initialization and in response to queries to oracle $\mathcal{O}\texttt{Progress}$ from adversary $\mathcal{A}$), adversary $\mathcal{B}_{\mathsf{OWC}}$ calls oracle $\mathcal{O}\texttt{New}$ to generate a new OWC instance in game $\mathbf{G}^{\text{ind-fs}}_{\mathsf{OWC},N}$ and stores the index of the new instance in $st.\mathcal{F}$ or $st.\mathcal{B}$ (instead of the OWC seed, which is unknown to adversary $\mathcal{B}_{\mathsf{OWC}}$). This implies that the states and extensions generated via oracles $\mathcal{O}\texttt{Export}$, $\mathcal{O}\texttt{CreateExt}$ and $\mathcal{O}\texttt{ProcExt}$ and stored in dictionaries $st.\mathcal{F}$, $st.\mathcal{B}$, $ext.\mathcal{F}$ and $ext.\mathcal{B}$ also contain indices in place of chain seeds. We will refer to these indices as "chain indices".

Second, when adversary $\mathcal{A}$ queries oracle $\mathcal{O}\texttt{GetKey}(e)$, adversary $\mathcal{B}_{\mathsf{OWC}}$ does not compute $\mathsf{OWC.GetKey}$ to generate fk and bk. (Cf. lines 46 and 48 in Figure 6.) Instead, $\mathcal{B}_{\mathsf{OWC}}$ issues two queries to its own $\mathcal{O}\texttt{GetKey}$ oracle; one for the forward chain and one for the backward chain, under the chain indices retrieved from $st.\mathcal{F}$ and $st.\mathcal{B}$ for the requested epoch e. Note that the epoch of the $\mathcal{O}\texttt{GetKey}$-calls by $\mathcal{B}_{\mathsf{OWC}}$ do not directly correspond to the DKR epoch e, but are computed relative to the starting epoch of the current chain. For example, if adversary $\mathcal{A}$ requests a key for epoch 15, and the forward

$\mathsf{GetFChainOWCEpoch}(\mathcal{F}, e^*)$:

$[(e_0, n_0), (e_1, n_1), \ldots, (e_{|\mathcal{F}|-1}, n_{|\mathcal{F}|-1})] \leftarrow \mathcal{F}$

let $i \in [0, |\mathcal{F}| - 1]$ s.t. $e_i \leq e^* < e_{i+1}$ for $e_{|\mathcal{F}|} = e_{|\mathcal{F}|-1} + N'$

return $(e^* - e_i, n_i)$

$\mathsf{GetBChainOWCEpoch}(\mathcal{B}, e^*)$:

$[(e_0, n_0), (e_1, n_1), \ldots, (e_{|\mathcal{B}|-1}, n_{|\mathcal{B}|-1})] \leftarrow \mathcal{B}$

let $i \in [0, |\mathcal{B}| - 1]$ s.t. $e_{i-1} < e^* \leq e_i$ for $e_{-1} = -1$

return $(e_i - e^*, n_i)$

Figure 20: Retrieve the OWC epoch index for the OWC chain on which the challenge epoch e^*, relative to the start of that chain.

chain used for this epoch starts at epoch 10 and has chain index 3 (i.e., it was generated via the third call to oracle $\mathcal{O}\texttt{New}$), then $\mathcal{B}_{\mathsf{OWC}}$ will query $\mathcal{O}\texttt{GetKey}(3, 5)$.

Next, when adversary $\mathcal{A}$ halts and outputs the challenge epoch e^* and requested list I of intervals (states) to expose, adversary $\mathcal{B}_{\mathsf{OWC}}$ does as follows. First, it recovers the relative challenge epochs and chain indices for the forward and backward chains by calling the helper functions $(e_f^*, n_f) \leftarrow \mathsf{GetFChainOWCEpoch}(st.\mathcal{F}, e^*)$ and $(e_b^*, n_b) \leftarrow \mathsf{GetBChainOWCEpoch}(st.\mathcal{B}, e^*)$ defined in Figure 20, respectively. Then, it checks if $\mathsf{SafeFC}(e^*)$ and $\mathsf{SafeBCAux}(e^*)$ (see bottom of Figure 19) are true for the challenge epoch e^*. If the $\mathsf{SafeFC}(e^*)$ is true (that is, the forward chain state at epoch e^* is not exposed by one of the intervals requested by $\mathcal{A}$), then $\mathcal{B}_{\mathsf{OWC}}$ sets $x_{n_f} \leftarrow (e_f^*, e_f^* + 1)$. Otherwise, it lets $x_{n_f} \leftarrow (-1, 0)$. Similarly for the backward chain, it sets $x_{n_b} \leftarrow (e_b^*, e_b^* + 1)$ if the backward chain is safe at epoch e^* (note that the seeds of the backward chain cannot be exposed via $\mathcal{O}\texttt{ExposeAux}$ as this violates the $\mathsf{SafeBCAux}(e^*)$ predicate), otherwise lets $x_{n_b} \leftarrow (-1, 0)$. For all other $n \in [1, m]$, where m is the total number of OWC instances generated by adversary $\mathcal{B}_{\mathsf{OWC}}$'s calls to oracle $\mathcal{O}\texttt{New}$, it lets $x_n \leftarrow (-1, 0)$. Adversary $\mathcal{B}_{\mathsf{OWC}}$ then halts and returns $(x_1, \ldots, x_{n_f}, \ldots, x_{n_b}, \ldots, x_m)$.[19] That is, for every OWC instance which does not correspond to the challenge forward or backward chain, as well as for the challenge chains if they are unsafe, $\mathcal{B}_{\mathsf{OWC}}$ does not request a challenge in the OWC game (indicated by a -1 as the challenge epoch) and instead asks to expose the seed (the OWC state with index 0). In return, it receives the seeds of all chains not covering the challenge epoch, as well as either

1. $(K_{n_f}^*, sst_{e_f^*+1}^{n_f})$ and $(K_{n_b}^*, sst_{e_b^*+1}^{n_b})$, if both chains are safe,
2. $(K_{n_f}^*, sst_{e_f^*+1}^{n_f})$ and $sst_0^{n_b}$, if only the forward chain is safe, or
3. $sst_0^{n_f}$ and $(K_{n_b}^*, sst_{e_b^*+1}^{n_b})$, if only the backward chain is safe.

Adversary $\mathcal{B}_{\mathsf{OWC}}$ uses the exposed seeds to compute the requested intervals for $\mathcal{A}$, as well as the backward and forward challenge key ($K_{n_b}^*$ and $K_{n_f}^*$) in case 2 and 3 above, respectively. It then proceeds as the challenger in game G_1, computes the challenge key as $k_{e^*} \leftarrow \mathsf{F}(K_{n_f}^*, K_{n_b}^*)$ and runs the second stage of adversary $\mathcal{A}$ on input k_{e^*} and the exposed states, again simulating queries to oracle $\mathcal{O}\texttt{GetKey}$, $\mathcal{O}\texttt{ExposeAux}$ (leaking the corresponding backward chain seeds) and the other oracles as above. When $\mathcal{A}$ halts and returns a bit guess b', $\mathcal{B}_{\mathsf{OWC}}$ also halts and returns b'. This way, adversary $\mathcal{B}_{\mathsf{OWC}}$ simulates game G_1 for $\mathcal{A}$ when it plays game $\mathbf{G}_{\mathsf{OWC},N}^{\text{ind-fs-1}}$ and game G_2 when it plays game $\mathbf{G}_{\mathsf{OWC},N}^{\text{ind-fs-0}}$. Note that by construction, adversary $\mathcal{B}_{\mathsf{OWC}}$ will not violate the trivial attack prevention mechanisms in the OWC game. (All of its challenge and expose requests are of the form $(-1, 0)$ or $(e, e + 1)$ for $e \in \mathbb{N}$.) Hence, we have $\Pr[\mathbf{G}_{\mathsf{OWC},N}^{\text{ind-fs-1}}(\mathcal{B}_{\mathsf{OWC}})] = \Pr[\mathrm{G}_1(\mathcal{A})]$ and

[19] The order depends on the values of n_f and n_b; x_{n_b} might appear before x_{n_f}.

$\Pr[\mathbf{G}^{\text{ind-fs-0}}_{\mathsf{OWC},N}(\mathcal{B}_{\mathsf{OWC}})] = \Pr[\mathrm{G}_2(\mathcal{A})]$. Together with the definition of ind-fs (see definition 3.1), this gives the bound in equation (9).

Adversary $\mathcal{B}_{\mathsf{OWC}}$ makes at most $2 \cdot \mathcal{Q}_{\mathcal{O}\texttt{GetKey}}(\mathcal{A})$ queries to oracle $\mathcal{O}\texttt{GetKey}$ and at most $2 + 2 \cdot \mathcal{Q}_{\mathcal{O}\texttt{Progress}}(\mathcal{A})$ queries to oracle $\mathcal{O}\texttt{New}$.

Double-PRF security of F. The last part of the proof consists of a reduction to the double-PRF security of F. To this end, we construct game G_3, shown in Figure 19. G_3 is identical to G_2, except that in the final computation of the challenge key, the call to F is replaced by random sampling from $\{0,1\}^\ell$. This means that game G_3 is equivalent to $\mathbf{G}^{\text{iac-0}}_{\mathsf{INT},\mathsf{Safe}}$, giving

$$\Pr[\mathbf{G}^{\text{iac-0}}_{\mathsf{INT},\mathsf{Safe}}(\mathcal{A})] = \Pr[\mathrm{G}_3(\mathcal{A})] \; . \tag{10}$$

We construct an adversary $\mathcal{B}_{\mathsf{F}}$ such that

$$\Pr[\mathrm{G}_2(\mathcal{A})] - \Pr[\mathrm{G}_3(\mathcal{A})] \leq \mathbf{Adv}^{\text{dbl-prf}}_{\mathsf{F}}(\mathcal{B}_{\mathsf{F}}) \; . \tag{11}$$

Adversary $\mathcal{B}_{\mathsf{F}}$ acts as the challenger in game G_2, except that instead of sampling fk and/or bk itself as on lines 56 and 57 of Figure 19, it calls oracle $\mathcal{O}\texttt{NewX}$ and $\mathcal{O}\texttt{NewY}$, respectively. Furthermore, instead of computing $k_{e^*} \leftarrow \mathsf{F}(fk, bk)$ as on line 58, adversary $\mathcal{B}_{\mathsf{F}}$ calls one of oracles $\mathcal{O}\texttt{FnX}(1, bk)$ (if only $\mathsf{SafeFC}(e^*)$ is true), $\mathcal{O}\texttt{FnY}(fk, 1)$ (if only $\mathsf{SafeBCAux}(e^*)$ is true) or $\mathcal{O}\texttt{FnXY}(1,1)$ (if both chains are safe at the challenge epoch) and uses the response as k_{e^*}. When adversary $\mathcal{A}$ halts and returns b', $\mathcal{B}_{\mathsf{F}}$ also halts and returns b'. Adversary $\mathcal{B}_{\mathsf{F}}$ hence makes at most one query to each of $\mathcal{O}\texttt{NewX}$ and $\mathcal{O}\texttt{NewY}$, as well as at most one query to either $\mathcal{O}\texttt{FnX}$, $\mathcal{O}\texttt{FnY}$ or $\mathcal{O}\texttt{FnXY}$. Furthermore, it perfectly simulates game G_2 resp. G_3 for $\mathcal{A}$ when bit b in $\mathbf{G}^{\text{dbl-prf-}b}_{\mathsf{INT}}$ is 1 and 0, respectively. This shows Equation (11).

Putting together Equations (7)–(11) gives the theorem statement. □

F Proof of Security for Grappa

F.1 Cleanness predicate C

We define our cleanness predicate $\mathsf{C} = \mathsf{C}(e^*)$ with respect to the challenge epoch e^* corresponding to the $\mathcal{O}\texttt{Test}$ query. The predicate models the security of Grappa when instantiated with the $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ interval scheme and with an optimally-secure CGKA such as RTreeKEM, following [6]. In a nutshell, the CGKA cleanness predicate $\mathsf{C}_{\mathsf{CGKA}}$ in [6] satisfies that $\mathsf{C}_{\mathsf{CGKA}} = \textbf{true}$ if, for every corruption, it holds that either: (a) the challenge epoch e^* occurs before the corruption (forward security), or (b) e^* is after a corruption, but the corrupted user updates or is removed before e^* (post-compromise security).

In Figure 21, we introduce C divided in several parts for easier readability. Essentially, as introduced below, the predicate closely reflects the steps that are taken in the security proof. The predicate is also able to express the best possible security that Grappa can achieve if it is instantiated with an interval scheme with perfect IAC. For this, one can simply replace the last line in Figure 21 by ($\textbf{return}\ b_L \wedge b_R$), and remove the predicates dedicated to derive $f_{\mapsto}$, f_0 and $f_{\leftarrow}$. Similarly, it is not hard to modify the predicate to reflect the security of Grappa if the interval scheme is instantiated differently, such as from GGM trees (see the discussion in Section 4.6).

In the cleanness predicate, IAC is strictly captured by allowing the adversary to test epoch keys from *any epoch for which none of the exposed users was a group member*. To see this, observe that, given a challenge epoch e^*, all flags $b_L, b_R, f_{\mapsto}, f_{\leftarrow}, f_0$ are set to **true** if all exposures correspond to parties that left the group before e^* or were added after e^*. PCS is also captured by considering

$\mathsf{C}(e^*)$:

// Testing a leaked epoch key not allowed
if $\exists q_i = \mathcal{O}\texttt{GetKey}(e^*)$ **return false**
// No exposures: all adversarial strategies allowed
if $\nexists q_i = \mathcal{O}\texttt{Expose}(\cdot,\cdot)$ **return true**
// Exposures on challenge epoch e^* not allowed
if $\exists q_i = \mathcal{O}\texttt{Expose}(\cdot, e^*)$ **return false**
// Admin Exposures: No admin is non-healed by e^*
for all $q_i = \mathcal{O}\texttt{Expose}(uid_i; e_i) : uid_i \in \boldsymbol{AD}[e_i]$
  if $\nexists e : e_i < e \leq e^*$ **and** e corresponds to $\mathcal{O}\texttt{UpdateAdm}(uid_i)$, $\mathcal{O}\texttt{RotKeys}(uid_i)$ or $\mathcal{O}\texttt{RemoveAdm}(\cdot, uid_i)$,
    return false
// Multi-window security: exposures both after and before e^*.
$Q_L \leftarrow \{q_i = \mathcal{O}\texttt{Expose}(uid_i, e_i) : e_i < e^*\}$ // exposures before e^*
$Q_R \leftarrow \{q_j = \mathcal{O}\texttt{Expose}(uid_j, e_j) : e_j > e^*\}$ // exposures after e^*
$b_L, b_R, f_{\mapsto}, f_0, f_{\leftarrow} \leftarrow$ **false**
// Exposures before e^*:
  if $\forall q_i \in Q_L$, $\exists e : e_i < e \leq e^*$ triggered by either any query made on $(uid_i, \cdot)$ if $uid_i \in \boldsymbol{AD}[e]$, an $\mathcal{O}\texttt{Remove}(\cdot, uid_i)$ query, or an $\mathcal{O}\texttt{Update}(uid_i)$ query:
    $b_L \leftarrow$ **true** // CGKA healing
    if $\forall q_i \in Q_L$, $\exists e' : e \leq e' \leq e^*$ triggered by an $\mathcal{O}\texttt{Remove}(\cdot,\cdot)$ query:
      $f_{\mapsto} \leftarrow$ **true** // forward flag exists
// Exposures after e^*:
  if $\forall q_j \in Q_R$, $\exists e : e^* < e \leq e_j$ **and** $uid_j \notin \boldsymbol{G}[e]$
    $b_R \leftarrow$ **true** // CGKA healing
    if $\forall q_j \in Q_L$, $\exists e' : e \leq e' \leq e^*$ triggered by an $\mathcal{O}\texttt{RotKeys}(\cdot,\cdot)$ or $\mathcal{O}\texttt{RemoveAdm}(\cdot,\cdot)$ query:
      $f_0 \leftarrow$ **true** // initial backward flag exists
    if $\forall q_j \in Q_R$, $\exists e' : e^* < e' < e_j$ **and** $uid_j \notin \boldsymbol{G}[e']$ **and** e' is triggered by an $\mathcal{O}\texttt{RotKeys}(\cdot,\cdot)$ or $\mathcal{O}\texttt{RemoveAdm}(\cdot,\cdot)$ query:
      $f_{\leftarrow} \leftarrow$ **true** // backward flag exists
return $b_L \wedge b_R \wedge (f_{\mapsto} \vee (f_{\leftarrow} \wedge f_0))$

Figure 21: Cleanness predicate $\mathsf{C} = \mathsf{C}(e^*)$ for the security of the GKP instantiation Grappa, given queries $q_1, \ldots, q_Q$ by the adversary.

that parties may heal after $\mathcal{O}\texttt{Update}$ and $\mathcal{O}\texttt{RotKeys}$ queries. The complexity of the predicate mostly arises from considering the case in which the adversary exposes multiple users with access to different epoch intervals. A lot of the complexity also stems from modelling admin corruptions; if these were fully disallowed, the predicate would also be much simpler.

F.2 Proof of Theorem 6.1

Let $q_1, \ldots, q_Q$ be the queries made by $\mathcal{A}$ to the different game oracles, which are announced in advance by the non-adaptive adversary. Let also q^* be the unique challenge query, $q^* = \mathcal{O}\texttt{Test}(e^*)$, where e^* is the *challenge epoch*. Let also e_{max} be the global epoch after query q_Q.

The proof proceeds via a series of game hops. We advance through the game hops as follows: if necessary, we first introduce a characterization of epochs or keys; then we introduce the next game hop, and finally we bound the difference in advantage in a lemma, that we prove.

To start, let $\mathrm{G}_0(\mathcal{A}) = \mathbf{G}^{\text{kind-1}}_{\mathsf{Grappa},\mathsf{C}}(\mathcal{A})$ be the real ($b = 1$) key indistinguishability game for GKP as in Figure 10. Given a sequence of adversarial queries, we define epoch e_{AD} as follows:

e_{AD}:	the smallest of all epochs e that satisfy: $\forall\ q_i = \mathcal{O}\texttt{Expose}(uid_i; e_i)$ such that $uid_i \in \boldsymbol{AD}[e_i]$ (i.e. uid_i is admin), $\exists e' : e_i < e' \leq e$ triggered by either: - an $\mathcal{O}\texttt{UpdateAdm}(uid_i)$ query, - an $\mathcal{O}\texttt{RotKeys}(uid_i)$ query, or - an $\mathcal{O}\texttt{RemoveAdm}(\cdot, uid_i)$ query.

In other words e_{AD} is the earliest epoch when the CGKA state of all admins that have ever been exposed by $\mathcal{A}$ may have healed. Note that interval scheme states may not be included here. To see why, observe that the CGKA cleanness predicate of the admin CGKA is $\mathsf{C}_{\mathsf{CGKA}}(e) = \textbf{true}$ for every $e \geq e_{\mathsf{AD}}$, but $\mathsf{C}_{\mathsf{CGKA}}(e_{\mathsf{AD}}) = \textbf{false}$. Note that if no epoch satisfies the condition, then $\mathsf{C} = \textbf{false}$ and hence the advantage of $\mathcal{A}$ is 0. Note also that we are referring to GKP epochs and not CGKA epochs; for the admin CGKA, multiple GKP epochs may map to a single CGKA epochs if no admin commands are executed.

Game 1. We define G_1 as G_0, except that all admin CGKA keys I_A corresponding to epochs in the interval $[e_{\mathsf{AD}}, e_{\mathsf{max}}]$ are replaced by uniformly random keys.

Lemma F.1. There exists an adversary $\mathcal{B}_{\mathsf{CGKA}}$ against CGKA kind security that runs in time similar to $\mathcal{A}$ and makes at most $3Q$ queries such that

$$\mathbf{Adv}^{\mathrm{G}_0}_{\mathsf{Grappa},\mathsf{C}}(\mathcal{A}) \leq \mathbf{Adv}^{\mathrm{G}_1}_{\mathsf{Grappa},\mathsf{C}}(\mathcal{A}) + Q \cdot \mathbf{Adv}^{\mathrm{kind}}_{\mathsf{CGKA}}(\mathcal{B}_{\mathsf{CGKA}}).$$

Proof. Between both games, we additionally define a series of hybrid games $\mathsf{Hyb}^{e_{\mathsf{AD}}-1}, \ldots, \mathsf{Hyb}^{e_{\mathsf{max}}}$ such that at Hyb^{i}, all admin CGKA keys in the interval $[e_{\mathsf{AD}}, i]$ are uniformly random keys. Note that $\mathsf{Hyb}^{e_{\mathsf{AD}}-1} = \mathrm{G}_0$ and $\mathsf{Hyb}^{e_{\mathsf{max}}} = \mathrm{G}_1$.

To transition throughout the hybrids, let $\mathcal{A}$ interpolate between Hyb^{j-1} and Hyb^{j} for some $j \in [e_{\mathsf{AD}}, e_{\mathsf{max}}]$. Then, we construct an adversary $\mathcal{B}_{\mathsf{CGKA}}$ against CGKA security that proceeds as below (we omit concrete query parameters for succinctness).

- $\mathcal{B}_{\mathsf{CGKA}}$ runs $\mathcal{A}$, receives the sequence of queries $q_1, \ldots, q_Q$ and initializes the GKP game dictionaries.
- For each oracle query made by $\mathcal{A}$, $\mathcal{B}_{\mathsf{CGKA}}$ simulates locally the code corresponding to all algorithms except for admin CGKA algorithms, for which it forwards the appropriate queries to the CGKA challenger.
- To simulate $\mathcal{O}\texttt{Expose}(uid)$ queries, the $\mathcal{B}_{\mathsf{CGKA}}$ also queries the $\mathcal{O}\texttt{Expose}(uid)$ CGKA oracle, and then forwards the CGKA state together with the rest of the simulated state.
- For the $\mathcal{O}\texttt{Test}$ and the $\mathcal{O}\texttt{GetKey}$ queries, $\mathcal{B}_{\mathsf{CGKA}}$ simulates the query internally and outputs the real K.
- Every time $\mathcal{B}_{\mathsf{CGKA}}$ needs to encrypt a ciphertext under I_A for some epoch $e \notin [e_{\mathsf{AD}}, j]$, it queries the CGKA $\mathcal{O}\texttt{GetKey}(e)$ oracle and uses the real key.
- For I_A keys of epochs $[e_{\mathsf{AD}}, j-1]$, it uses a (consistent) uniformly random key.
- For I_A at epoch j, it queries the $\mathcal{O}\texttt{Test}$ oracle of the CGKA challenger and uses the key sent by challenger.
- At the end of the game, $\mathcal{B}_{\mathsf{CGKA}}$ outputs whatever $\mathcal{A}$ outputs.

Note that the cleanness predicate $\mathsf{C}_{\mathsf{CGKA}} = \textbf{true}$, as (1) the CGKA challenge key (j-th epoch) is never queried to $\mathcal{O}\texttt{GetKey}$, and (2) all exposures occur such that all epochs $e \geq e_{\mathsf{AD}}$ are safe (except for individual $\mathcal{O}\texttt{GetKey}$ queries). As $\mathcal{B}_{\mathsf{CGKA}}$ perfectly simulates the games for $\mathcal{A}$, we have that

$$\Pr[\mathsf{Hyb}^{j-1}_{\mathsf{Grappa},\mathsf{C}}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathsf{Hyb}^{j}_{\mathsf{Grappa},\mathsf{C}}(\mathcal{A}) \Rightarrow 1] \leq$$
$$\Pr[\mathbf{G}^{\text{kind-1}}_{\mathsf{CGKA},\mathsf{C}_{\mathsf{CGKA}}}(\mathcal{B}_{\mathsf{CGKA}}) \Rightarrow 1] - \Pr[\mathbf{G}^{\text{kind-0}}_{\mathsf{CGKA},\mathsf{C}_{\mathsf{CGKA}}}(\mathcal{B}_{\mathsf{CGKA}}) \Rightarrow 1] = \mathbf{Adv}^{\text{kind}}_{\mathsf{CGKA}}(\mathcal{B}_{\mathsf{CGKA}})$$

and the lemma follows by aggregating the bounds for all (at most Q) hybrids. Furthermore, $\mathcal{B}_{\mathsf{CGKA}}$ makes at most three queries to its CGKA challenger per each query of $\mathcal{A}$, and so it makes at most $3Q$ queries in total. □

Game 2. We define G_2 as G_1, except that all ciphertexts C_A corresponding to epochs in the interval $[e_{\mathsf{AD}}, e_{\mathsf{max}}]$ are replaced by random strings of the same length.

Lemma F.2. There exists an adversary $\mathcal{B}_{\mathsf{AEAD}}$ against ind$-cpa security of AEAD that runs in time similar to $\mathcal{A}$ and makes at most Q queries such that

$$\mathbf{Adv}^{\mathrm{G}_1}_{\mathsf{Grappa},\mathsf{C}}(\mathcal{A}) \leq \mathbf{Adv}^{\mathrm{G}_2}_{\mathsf{Grappa},\mathsf{C}}(\mathcal{A}) + \mathbf{Adv}^{\text{ind\$-cpa}}_{\mathsf{AEAD}}(\mathcal{B}_{\mathsf{AEAD}}).$$

Proof. All the ciphertexts that have been replaced in the game were encrypted using uniformly random keys, since these are exactly the I_A keys from epochs $[e_{\mathsf{AD}}, e_{\mathsf{max}}]$. Hence, the lemma follows by a standard reduction to ind$-cpa security of the AEAD scheme (Definition B.3). We omit the details. □

Characterization of safety window. Before moving on to the next hybrid, we characterize an interval where we will replace member CGKA keys by random, that we denote by *safety window* **SW**. Let q^* be the challenge query and e^* the challenge query as before. We define e_L to be the latest epoch before e^* such that a $\mathcal{O}\texttt{Expose}(\cdot)$ query is made; if no exposures before e^* occur, then we let $e_L \leftarrow -1$.

We also define e_R to be the earliest epoch after e^* for which all users that were compromised after the challenge were still not group members. In other words e_R captures interval access security for epochs in the past, and we define it formally below. Finally, we define a healing epoch $e_{\mathsf{PCS}} < e^*$ as follows. Clearly, if $e_L = -1$ then $e_{\mathsf{PCS}} = 0$:

$\underline{e_R}$:	Define e_i for each uid_i as the latest epoch such that: $q_j = \mathcal{O}\texttt{Expose}(uid_i; e_j)$ occurred for some $e_j \geq e^*$, $uid_i \notin \boldsymbol{G}[e_i]$ (i.e. uid_i was not a group member at e_i). Then, $e_R \leftarrow \min_i e_i$. If no such q_j exist, $e_R \leftarrow e_{\mathsf{max}} + 1$.
$\underline{e_{\mathsf{PCS}}}$:	the smallest of all epochs e that satisfy: $\forall\ q_i = \mathcal{O}\texttt{Expose}(uid_i; e_i)$ such that $uid_i \in \boldsymbol{G}[e_i]$ and $e_i < e^*$, $\exists e' : e_i < e' \leq e$ triggered by either: - any query made on $(uid_i, \cdot)$ if $uid_i \in \boldsymbol{AD}[e']$, - an $\mathcal{O}\texttt{Remove}(\cdot, uid_i)$ query, or - an $\mathcal{O}\texttt{Update}(uid_i)$ query.

Claim F.3. If $e_R \leq e^*$ or if e_{PCS} does not exist, then $\mathsf{C} = \textbf{false}$. Moreover, for the member CGKA, we have that all epochs $e \in [e_{\mathsf{PCS}}, e_R]$ satisfy that $\mathsf{C}_{\mathsf{CGKA}}(e) = \textbf{true}$.

Proof. The claim follows by inspection of Figure 21. If $e_R \leq e^*$, the bit b_R in the description of C remains **false**. Similarly, if e_{PCS} does not exist, we have that b_L also remains **false**.

For the second part, note that: (a) no corruptions occur in the interval $[e_{\mathsf{PCS}}, e_R]$, (b) the member CGKA is healed at e_{PCS} (and therefore also at e_R) since the corrupted uid_i has been removed or updated its CGKA keys, and (c) by the optimal forward security, e_R is still safe regardless of corruptions that occur in posterior epochs. □

Finally, we let $\mathbf{SW} = [e_{\mathsf{PCS}}, e_R]$. By noting that $e_L \neq e^*$ and $e_R \neq e^*$ (as otherwise C is trivially false), we have that $e_L < e_{\mathsf{PCS}} \leq e^* < e_R$, and it follows that $\mathbf{SW} \neq \emptyset$.

Game 3. We define G_3 as G_2, except that all member CGKA keys I_M corresponding to epochs in the interval $\mathbf{SW}$ are replaced by uniformly random keys.

Lemma F.4. There exists an adversary $\mathcal{B}_{\mathsf{CGKA}}$ against CGKA kind security that runs in time similar to $\mathcal{A}$ and makes at most $3Q$ queries such that

$$\mathbf{Adv}^{\mathrm{G}_2}_{\mathsf{Grappa},\mathsf{C}}(\mathcal{A}) \leq \mathbf{Adv}^{\mathrm{G}_3}_{\mathsf{Grappa},\mathsf{C}}(\mathcal{A}) + Q \cdot \mathbf{Adv}^{\mathrm{kind}}_{\mathsf{CGKA}}(\mathcal{B}_{\mathsf{CGKA}}).$$

Proof. The proof follows a similar strategy as the proof of Lemma F.1. We define a sequence of hybrid games $\mathsf{Hyb}^{e_{\mathsf{PCS}}-1}, \ldots, \mathsf{Hyb}^{e_R}$ such that at Hyb^i, all admin CGKA keys in the interval $[e_{\mathsf{PCS}}, i]$ are uniformly random keys. As before, note that $\mathsf{Hyb}^{e_{\mathsf{PCS}}-1} = \mathrm{G}_3$ and $\mathsf{Hyb}^{e_R} = \mathrm{G}_4$.

To transition throughout the hybrids, we carry out a simulation strategy exactly as in the proof of Lemma F.1, except that instead of simulating the admin CGKA, we simulate the member CGKA operations depending on the adversary's oracle calls. Due to the similarity of both proofs, we skip further details. □

Game 4. We define G_4 as G_3, except that all ciphertexts C_A corresponding to epochs in $\mathbf{SW}$ are replaced by random strings of the same length.

Lemma F.5. There exists an adversary $\mathcal{B}_{\mathsf{AEAD}}$ against ind$-cpa security of AEAD that runs in time similar to $\mathcal{A}$ and makes at most Q queries such that

$$\mathbf{Adv}^{\mathrm{G}_3}_{\mathsf{Grappa},\mathsf{C}}(\mathcal{A}) \leq \mathbf{Adv}^{\mathrm{G}_4}_{\mathsf{Grappa},\mathsf{C}}(\mathcal{A}) + \mathbf{Adv}^{\text{ind\$-cpa}}_{\mathsf{AEAD}}(\mathcal{B}_{\mathsf{AEAD}}).$$

Proof. The proof is identical to the proof of Lemma F.2, so we omit the details. □

Progress flags in the safety window. For the last step of the proof, we need to characterize the epochs where a healing of the interval scheme may have occured. Indeed, in the safety window $\mathbf{SW}$ there are no current exposures regarding CGKA material, but there may be past (or future) exposures cause leakage at the interval scheme level. We will define a flag window $\mathbf{FW}$ that captures the epochs where the interval scheme may have been compromised. We remark that if the interval scheme had perfect IAC (i.e. $\mathsf{Safe} = \mathbf{true}$), then at this step we could set $\mathbf{FW} = \mathbf{SW}$. Unfortunately, the non-trivial safety predicate of $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ makes the following proof inevitably cumbersome.

We define epochs $e_{\leftarrow\!|}, e_0, e_{|\!\rightarrow} \in \mathbf{SW}$ as follows[20]:

[20] For the rest of the proof, we will often refer to interval scheme epochs, which may differ from GKP epochs. The cause of the mismatch is that the $\mathcal{O}\mathtt{Update}$ oracle does not progress the interval scheme global state of Grappa. interval scheme epochs e_{INT} can be computed as $e_{\mathsf{INT}} = e_{\mathsf{GKP}} - \mathsf{up}[e_{\mathsf{GKP}}]$ where $\mathsf{up}[e_{\mathsf{GKP}}]$ is the value of the game variable up at the time of e_{GKP}. We omit refering to these explicitly in order to avoid increasing the (already notable) notational complexity in the proof.

$\underline{e_{\leftarrow\!\mid}}$:	earliest epoch $e \in [e^* + 1, e_R]$ triggered by either: - an $\mathcal{O}\texttt{RemoveAdm}(\cdot,\cdot)$ query, or - an $\mathcal{O}\texttt{RotKeys}(\cdot)$ query.
$\underline{e_0}$:	latest epoch $e \in [e_{\mathsf{PCS}}, e^*]$ triggered by either: - an $\mathcal{O}\texttt{RemoveAdm}(\cdot,\cdot)$ query, or - an $\mathcal{O}\texttt{RotKeys}(\cdot)$ query.
$\underline{e_{\mapsto\!\mid}}$:	latest epoch $e \in [e_{\mathsf{PCS}}, e^*]$ triggered by: - an $\mathcal{O}\texttt{Remove}(\cdot,\cdot)$ query.

Claim F.6. If $\mathsf{C} = \textbf{true}$, then either $e_{\mapsto\!\mid} \neq \bot$, or both $e_0, e_{\leftarrow\!\mid} \neq \bot$

Proof. By inspection of Figure 21, observe that if $e_{\mapsto\!\mid} = \bot$ then $f_{\mapsto\!\mid}$ is also **false**. This is apparent in line 13 of C, where the condition is that $\exists e' : e \leq e' \leq e^*$ triggered by an $\mathcal{O}\texttt{Remove}$ query, and where $e \geq e_{\mathsf{PCS}}$ is in the safety window **SW** by definition of the latter. Similarly, if $e_{\leftarrow\!\mid} = \bot$ then $f_{\leftarrow\!\mid}$ is also **false**, and if e_0 then f_0 is **false**. The argument follows by looking at lines 19 and 17 of C. For f_0, the argument is identical to the one above. For $f_{\leftarrow\!\mid}$, we require the condition that $uid_j \notin \boldsymbol{G}[e']$ to be true, which is met as $e' \leq e_R$. The claim follows as to make $\mathsf{C} = \textbf{true}$ we need either $f_{\mapsto\!\mid}$ or both $f_{\leftarrow\!\mid}$ and f_0 to be true. □

We also define $\textbf{EXP} = \{e : \exists q_i = \mathcal{O}\texttt{GetKey}(e)\}$ as the set of all epochs whose key has been exposed via the $\mathcal{O}\texttt{GetKey}(e)$ oracle. Finally, we define a flag window **FW** as follows:

$$\textbf{FW} = \begin{cases} [e_{\mapsto\!\mid}, e_R] & \text{if} \quad e_0, e_{\leftarrow\!\mid} = \bot \\ [e_0, e_{\leftarrow\!\mid}] & \text{if} \quad e_{\mapsto\!\mid} = \bot \\ [\max(e_0, e_{\mapsto\!\mid}), e_{\leftarrow\!\mid}] & \text{if} \quad e_0, e_{\leftarrow\!\mid}, e_{\mapsto\!\mid} \neq \bot \end{cases}.$$

The intuition behind the definition of **FW** is as follows. Our goal is that either the backward or the forward chain are safe in the flag window. For the backward chain to be safe, we need that there is a backward chain rotation at e_0 after e_{PCS}. This will be sufficient as all admins recover from state exposure at $e_{\mathsf{AD}} \leq e_{\mathsf{PCS}}$. Additionally, we need that there is another rotation after e^* but before e_R, i.e., at $e_{\leftarrow\!\mid}$, as it gets compromised at that point in time by definition of e_R. For the forward chain to be safe, we simply need that there is a forward chain rotation at $e_{\mapsto\!\mid}$: after e_{PCS} but before the challenge epoch.

Claim F.7. The following are true about **FW**: (a) All epochs $e \in \textbf{FW} \setminus \textbf{EXP}$ satisfy that $\mathsf{Safe}(e) = \textbf{true}$, (b) $e^* \in \textbf{FW}$, and (c) there are no chain rotations at any $e \in \textbf{FW}$.

Proof. The first two claims follow from the definition of the safety predicate Safe for $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ in Figure 8. For (a), first note that **EXP** is equivalent to $\boldsymbol{X}_k$ at the interval scheme level. Second, note that if $e_{\mapsto\!\mid}$ exists, then the SafeFC part of Safe is true. Third, if $e_{\leftarrow\!\mid}$ exists, then SafeBC is also true. Furthermore, if e_0 exists, then $\mathsf{SafeAux}$ is also true, as all the exposures that may leak *aux* values (i.e., admin exposures) are healed before e_{PCS} and do never occur after e^* (by the definition of e_{AD}).

For (b), note that by definition $e_{\mathsf{PCS}} \leq e_{\mapsto\!\mid}, e_0 < e^* < e_{\leftarrow\!\mid} \leq e_R$. Finally, for (c) note that the only operations that may trigger a chain rotation are $\mathcal{O}\texttt{RemoveAdm}(\cdot,\cdot)$, $\mathcal{O}\texttt{RotKeys}(\cdot)$ and $\mathcal{O}\texttt{Remove}(\cdot,\cdot)$ queries, and the boundaries of **FW** are defined such that these queries do not occur inside the window. □

Game 5. We define G_5 as G_4, except that all interval scheme epoch keys k of the epochs $e \in \textbf{FW}$ are replaced by uniformly random keys. This includes the challenge epoch e^*.

Lemma F.8. There exists an adversary $\mathcal{B}_{\mathsf{INT}}$ against kind security of the interval scheme $\mathsf{D}[\mathsf{OWC}, \mathsf{F}]$ that runs in time similar to $\mathcal{A}$ and makes at most Q queries such that

$$\mathbf{Adv}^{\mathrm{G}_4}_{\mathsf{Grappa},\mathsf{C}}(\mathcal{A}) \leq \mathbf{Adv}^{\mathrm{G}_5}_{\mathsf{Grappa},\mathsf{C}}(\mathcal{A}) + \mathbf{Adv}^{\mathrm{iac}}_{\mathsf{INT}}(\mathcal{B}_{\mathsf{INT}}).$$

Proof. We construct $\mathcal{B}_{\mathsf{INT}}$ from $\mathcal{A}$ as follows. First, $\mathcal{B}_{\mathsf{INT}}$ initializes $\mathcal{A}$ and obtains the sequence of queries $q_1, \ldots, q_Q$ from $\mathcal{A}$. Then, $\mathcal{B}_{\mathsf{INT}}$ interacts with the oracles in the interval scheme IAC security game as a function of the queries of $\mathcal{A}$. Namely, for $i = 1, \ldots, Q$, $\mathcal{B}_{\mathsf{INT}}$ makes the following oracle calls, matching the Grappa algorithms:

- If q_i is the $\mathcal{O}\mathtt{Test}$ query or an $\mathcal{O}\mathtt{Init}$ query, ignore.
- If q_i is different than $\mathcal{O}\mathtt{Expose}$ and $\mathcal{O}\mathtt{Deliver}$, query $\mathcal{O}\mathtt{Progress}$ with the appropriate flag, and then $\mathcal{O}\mathtt{CreateExt}(\mathsf{ep}[q_i], 1)$ where ep is the global epoch game variable.
- If q_i is a $\mathcal{O}\mathtt{Add}$ query, call $\mathcal{O}\mathtt{Export}(\mathsf{ep}[q_i] + 1, \mathsf{ep}[q_i] + 1)$
- If q_i is a $\mathcal{O}\mathtt{AddAdm}$ query, call $\mathcal{O}\mathtt{Export}(1, \mathsf{ep}[q_i] + 1)$.
- If q_i is a $\mathcal{O}\mathtt{Deliver}(\mathit{uid}, e)$ query, call $\mathcal{O}\mathtt{ProcExt}(i_{st}, i_{ext})$ where the indices correspond to the state of uid on epoch e and the extension in $\boldsymbol{T}[e]$, respectively.
- If q_i is a $\mathcal{O}\mathtt{GetKey}(e)$ query, call the interval scheme oracle $k_e \leftarrow \mathcal{O}\mathtt{GetKey}(e)$.
- If q_i is a $\mathcal{O}\mathtt{Expose}(\mathit{uid}, e)$ query, ignore (for now) the query.

After this first round of queries to the interval scheme challenger, $\mathcal{B}_{\mathsf{INT}}$ defines an exposure set $\mathbf{EI} \leftarrow [0, \mathsf{e_{max}}] \setminus \mathbf{FW}$, and queries the $\mathcal{O}\mathtt{Export}$ oracle accordingly to obtain two exposure interval indices, denoted by i_1 and i_2, which span exactly the interval scheme epochs in $\mathbf{EI}$.

Finally, $\mathcal{B}_{\mathsf{INT}}$ announces an interval scheme challenge epoch e^*[21]. and exposure intervals $I = \{i_1, i_2\}$. Then, the $\mathcal{B}_{\mathsf{INT}}$ obtains access to interval states corresponding to all epochs $e \in \mathbf{EI} \cup \mathbf{EXP}$ (note that all keys in $\mathbf{EXP}$ were already queried during the first round of queries), as well as a challenge key k^*. Without loss of generality, we assume that the interval states contain the information required to recreate a (partial) global state that allows $\mathcal{B}_{\mathsf{INT}}$ to replicate the creation of interval states within $\mathbf{EI}$, as well as their corresponding extensions.

Next, $\mathcal{B}_{\mathsf{INT}}$ simulates to $\mathcal{A}$ as follows:

- For all oracle queries that correspond to epochs $e \notin \mathbf{FW}$, $\mathcal{B}_{\mathsf{INT}}$ runs the Grappa protocol for $\mathcal{A}$ identically as the challenger in G_4, generating the interval scheme states and extensions on-demand. Upon $\mathcal{O}\mathtt{Expose}$ oracles such that $\boldsymbol{AD}[\mathit{uid}; e] = 1$, i.e., corresponding to current admins, $\mathcal{B}_{\mathsf{INT}}$ also queries the interval scheme oracle $\mathcal{O}\mathtt{ExposeAux}(e)$ to obtain the backward chain seed and send it to $\mathcal{A}$ as part of the leaked admin state.
- For all oracle queries corresponding to epochs $e \in \mathbf{FW}$, $\mathcal{B}_{\mathsf{INT}}$ also simulates as the challenger in G_4, except that all interval scheme internal states and extensions are replaced by random strings. Recall that no $\mathcal{O}\mathtt{Expose}$ queries are made in $\mathbf{FW}$.
- For oracle queries $\mathcal{O}\mathtt{GetKey}(e)$, reply with the real epoch key, given that $e \in \mathbf{EXP}$.
- For the oracle query $q^* = \mathcal{O}\mathtt{Test}(e^*)$, $\mathcal{B}_{\mathsf{INT}}$ simply forwards k^* to $\mathcal{A}$ in G_4, and forwards a uniformly random key $k_0 \leftarrow\!\!\$\ \{0,1\}^\ell$ in G_5.

[21] following the previous footnote, this is the interval scheme epoch corresponding to e^* in the GKP game

Finally, $\mathcal{B}_{\mathsf{INT}}$ outputs whatever $\mathcal{A}$ outputs. As all ciphertexts corresponding to epochs $e \in \mathbf{FW} \subseteq \mathbf{SW}$ were already randomized in G_4, the games are identical in the view of $\mathcal{A}$ except for the $\mathcal{O}\mathtt{Test}$ query. As we argued before, $\mathsf{Safe}(e^*) = \mathbf{true}$, $\mathcal{B}_{\mathsf{INT}}$ is a valid adversary against the interval scheme, and the lemma follows. □

Conclusion. In game G_5, we have that $\mathbf{Adv}^{\mathrm{G}_5}_{\mathsf{Grappa},\mathsf{C}}(\mathcal{A}) = 0$, as the challenge key is replaced by a uniformly random key. Hence, G_5 is equivalent to the random ($b = 0$) GKP game $\mathbf{G}^{\text{kind-0}}_{\mathsf{Grappa},\mathsf{C}}$. By collecting the advantages from the lemmas, the theorem follows.