Fork-Resilient Continuous Group Key Agreement

Joël Alwen[1] Marta Mularczyk[1] Yiannis Tselekounis[2]

[1]AWS-Wickr, {jalwen,mulmarta}@amazon.ch
[2]Carnegie Mellon University, itseleko@cs.cmu.edu

March 19, 2023

Abstract. Continuous Group Key Agreement (CGKA) lets a evolving group of clients agree on a sequence of group keys. An important application of CGKA is scalable asynchronous end-to-end (E2E) encrypted group messaging.

A major problem preventing the use of CGKA over unreliable infrastructure are so-called forks. A *fork* occurs when group members have diverging views of the group's history (and thus its current state); e.g. due to network or server failures. Once communication channels are restored, members *resolve* a fork by agreeing on the state of the group again. Today's CGKA protocols make fork resolution challenging, as natural resolution strategies seem to conflict with the way the protocols enforce group state agreement and forward secrecy. Meanwhile, secure group messaging protocols which do support fork resolution do not scale nearly as well as CGKA does.

In this work, we pave the way to practical scalable E2E messaging over unreliable infrastructure. To that end, we generalize CGKA to *Fork Resilient*-CGKA which allows clients to process significantly more types of out-of-order network traffic. This is important for many natural fork resolution procedures as they are based, in part, on replaying missed traffic. Next, we give two FR-CGKA constructions: a practical one based on the CGKA underlying the MLS messaging standard and an optimally secure one (albeit with only theoretical efficiency). To further assist with fork resolution, we introduce a simple new abstraction to describe a client's local protocol state. The abstraction describes all and only the information relevant to natural fork resolution, making it easier for higher-level fork resolution procedures to work with and reason about. We define a black-box extension of an FR-CGKA which maintains such a description of a client's internal state. Finally, as a proof of concept, we give a basic fork resolution protocol.

Contents

1 Introduction **3**
1.1 Our Contribution . . . 6
1.2 Related Work . . . 8

2 Preliminaries **9**
2.1 Notation . . . 9
2.2 Binary-Tree Encryption and HIBE . . . 9
2.3 Binary-Tree Signatures . . . 10
2.4 Collision-Resistant PRF . . . 11
2.5 Puncturable PRF . . . 11

3 Fork-Resilient CGKA **13**
3.1 (Server-aided) CGKA . . . 13
3.2 FR-CGKA Protocols . . . 14
3.3 FR-CGKA Security Definition . . . 15
3.4 (Sub-)Optimal Security Predicates . . . 19

4 The FREEK Protocol **20**
4.1 Definition of FREEK . . . 23

5 FR-CGKA with Optimal Security **25**
5.1 Authenticity . . . 27
5.2 The Final Construction . . . 28

6 Natural Fork-Resolution Protocols **29**

7 Security of FREEK **32**
7.1 Security Proof of FREEK . . . 33
7.2 Consistency and Step Correctness . . . 36
7.3 Confidentiality . . . 39
7.4 Authenticity . . . 45

8 Security Proof of O-FREEK **48**
8.1 Consistency and Step Correctness . . . 49
8.2 Confidentiality . . . 50
8.3 Authenticity . . . 53

A Security Model Intuition **59**

B Details of SAIK Components Used by FREEK **61**

1 Introduction

End-to-end (E2E) encrypted secure messaging is a widely used class of cryptographic applications allowing groups of clients to communicate securely with each other over untrusted network and server infrastructure. Here, "secure" has come to denote (at least) message authenticity and a strong flavour of message confidentiality known as *Post Compromise Forward Secrecy* (PCFS) [CCG16, ACDT20]. Intuitively, PCFS means that current messages remain semantically secure from an adversary that controls all network traffic and can leak all participants' local states both in the past (PCS) and future (FS) but not currently.

A new class of messaging applications are based on underlying Continuous Group Key Agreement (CGKA) protocols, including the IETF's upcoming *Messaging Layer Security* (MLS) standard [BMO$^+$18]. Most of the functionality, security and efficiency properties of these protocols is inherited directly from their underlying CGKAs [ACDT21]. This has made CGKA a growing subject of cryptographic research in recent years. (See the related work section below for some highlights.)

Put simply, CGKA provides an E2E secure group state management for dynamic groups, i.e., groups whose properties (including its membership) evolve over time. Every change of the group's properties initiates a new *epoch*. The goal of CGKA is to equip the group members in each epoch with a symmetric *group key* known only to them. In line with the E2E security paradigm, clients can affect and authenticate changes to their group's state (including adding/removing members) on their own. That is, without relying on specially designated group members or trusted third parties (beyond some form of PKI). Finally, as sessions may last for years, at any point clients can perform a *PCS update*. This injects fresh entropy into all of the client's cryptographic secrets related to the CGKA session so as to hedge against the client's local state being leaked mid-session.

To be useful for messaging applications, CGKAs must provide all group members with a *consistent* view of the group's state. That is, group members that have a common view of a group state and subsequently see the same[1] "packets" (aka. protocol messages) will end up with the same new group state. This state includes the group's current membership and other application-relevant data (e.g. the group's name).

The CGKA-based approach to group messaging comes with various improvements over previous generations of messaging protocols. Most notably, CGKAs are designed to scale to orders of magnitude larger group sizes n (e.g. $100x$ larger) opening up new use cases in the domain of IoT and large scale events while reducing the bandwidth requirements of today's use cases. To enable this scalability, they reduce the $O(n)$ communication (and computational) complexities of older messaging protocols' (e.g. when removing a group member or doing a PCS update) [Fou23a, MP22, Wha23, HLA19] down to $O(\log(n))$; albeit, under some relatively mild assumptions about clients online/offline behavior. This type of complexity is sometimes informally referred to as "fair-weather complexity" and it is a hallmark of CGKA protocols. Other improvements of CGKA over older protocols include mechanisms for exporting shared keys (for use by higher level applications), importing PSKs and other arbitrary external contexts, dynamically updating ciphersuites, client capabilities and even the protocol version mid-session and a mechanism for recovering from faults. They are also very extensible allowing applications to define custom group properties, business logic, and protocol functionality as needed.

Consistency For Reliable Infrastructure. One of the main tasks in designing a messaging system (and CGKA) is to ensure all participants maintain a *consistent* view of the group's state under minimal network assumptions. Indeed, the vast majority of messaging applications are designed for asynchronous communication. That is, they allow clients to participate in groups even when no other

[1]or equivalent, as defined by the protocol

group member is currently online. Whenever a client does come online it quickly catches up to the current group state by processing any packets it missed while offline.

A common method of ensuring consistency is to implement a single source of truth determining the group state; e.g. via an (untrusted) server which buffers all packets for a group and forwards them *in the same order* to all group members as they come online. This implicitly establishes a fixed order of events in the group that all clients can agree on and based on which they can determine the new group state.[2] Alternatively, the server may even maintain its own view of the group state (possibly obliviously so [CPZ20]) which serves as a more direct single source of truth.

Consistency for Unreliable Infrastructure. Unfortunately, when infrastructure such as the network or server are *unreliable*, depending so critically on a central server can become problematic as it introduces a single point of failure (and a bottleneck) for the group. A client that cannot reach the server cannot participate in the group at all. Unfortunately (and unnecessarily as it turns out) that is even true when clients can still reach each other somehow over the network.

Yet many secure messaging use cases must contend with unreliable infrastructure. For example, in disaster relief scenarios, when communicating via ad-hoc mesh networks or when operating in contested environments. Even in a federated setting (where clients' packets are routed via host servers as with email) it can happen that host servers lose connectivity with each other for extended periods of time.[3] In such cases, we could ask that clients can still participate in groups with each other as long as their host servers remain in contact (or they are hosted on the same server).

Fortunately, as a consequence of enabling asynchronous messaging, most CGKA and secure messaging protocols already, in principle, allow arbitrary subsets of clients in a group to process each other's packets without any further interaction with the group members outside their subset [MP16, Fou23a, HLA19, BMO$^+$18]. Thus, one way to improve resilience of a messaging application is for clients to forward packets directly to each other whenever possible as permitted by the communication infrastructure.

But forgoing a central server also means living without the single source of truth it provided. So, to guarantee consistency a new method is needed that allows for "fork resolution". A *fork* occurs when clients in a group have diverging views of the group's event history, e.g. due to network links going down for some time. Once connectivity is restored, the fork resolution method must provide a way for clients to reconcile their divergent views in order to agree on a new group state from which to proceed.

For example, consider a group in a federated setting where some members are hosted on a host server S_A and the rest on host server S_B. Now suppose the link connecting the two servers goes down, partitioning the group into two subsets that can continue to communicate with each other via their shared host servers. Alice, who is hosted on server S_A adds Anthony to the group who joins from the perspective of all clients hosted on S_A. Meanwhile Bob, who is hosted on S_B, removes Alice from the group. Thus we now have a fork. Later, the link between the servers is restored. A fork resolution method must now provide a way for all clients in the group to resolve their fork to reach consensus on a common view of the group state (in particular, in this case, its membership).

[2]We note such delivery servers are not trusted for confidentiality, authenticity or agreement of the CGKA / messaging application. Instead, we rely on them only for availability. The agreement property ensures that for two clients to be in the same epoch (a prerequisite for exchanging E2E encrypted messages in a CGKA-based messaging protocol) the clients must first have the same view of the group state.

[3]Federated messaging is used widely in practice, especially in the enterprise and public sectors. [OR93, Jab23, Fou23a, Tea23, HLA19, Gmb21]. One reason is that by administering their own host servers, organizations can better manage their members' clients. For example, organizations can better control incoming/ougoing communication flows by determining to which external host servers their own server can connect.

Natural Fork Resolution. Fork resolution for CGKA and messaging have both been considered before. To the best of our knowledge, they all adhere to the following high-level outline we shall call *natural fork resolution.* First, clients determine which network packets each one is missing. Next, they obtain those packets (e.g. from each other via a gossip protocol). Finally, after processing the new packets they determine a new group state based on their, now updated, view of all events in the group. This final step is often implemented non-interactively by running an algorithm (sometimes called a "state resolution" algorithm [Fou23b]) which maps an initial group state and a set of causally dependent events in the group to a new group state.

The Matrix messaging application, the DCGKA protocol of [WKHB21] and the messaging protocol of [CEST22] employ the natural fork resolution paradigm. Along with a state resolution algorithm, each also introduces a group messaging protocol that allows processing packets delivered in any causality respecting order.

An event E_1 is *causally dependent* on event E_0 if E_1 happened after E_0 from the point of view of the client that created event E_1 [Wei19]. For example Alice must join the group (E_0) before she sends a message to the group (E_1). In a slight abuse of notation we use the same terminology for packets to denote that the sender of a packet P_1 already received packet P_0 when sending P_1. A sequence of events (or packets) is *causality respecting* if every event (or packet) appears in the sequence after all events (or packets) upon which it is causally dependent. Note that an otherwise unreliable network that does ensure packets from the same origin to the same destination are received in the order they are sent guarantees that packets in a session are delivered in an arbitrary but causality respecting order to all group members.[4]

However, neither Matrix nor the protocols from [WKHB21, CEST22] are designed with the type of scalability that characterizes CGKA protocols. In particular, rather than fair-weather logarithmic complexity, removing a group member requires $O(n)$ download bandwidth for each client, as does a single PCS update.[5] Conversely, the (experimental) fair-weather dMLS messaging protocol (adapted from MLS by the Matrix team [Mat23a]) allows for processing packets in any causality respecting order. However it does so at the cost of seriously weakening forward secrecy for message confidentiality.

Nevertheless, their state resolution algorithms (or something similar) seem like useful building blocks for a fork resilient fair weather complexity messaging protocol. But before we can apply them we must first overcome a fundamental problem with today's CGKA protocols. Forks can result in sender and receiver of a packet receiving packets in different orders. Thus, the natural fork resolution paradigm requires that clients can process incoming packets in arbitrary causally respecting order while ending up with the same interpretation of events encoded in those packets.

In the example above, clients hosted by server S_A will see Alice's packet inviting Anthony before they see Bob's packet removing Alice while clients hosted by server S_B will see the packets in reverse. Yet, to date, no CGKA supports this level of flexibility. Instead, to ensure forward secrecy of old epoch keys, protocols mandate critical key material in an epoch state be deleted as soon as it is used to transition to a new epoch. The unfortunate side effect is that no other transitions (e.g. due to forks) can be made out of the old epoch. Thus, clients are effectively only able to process events in a single sequence, making consistency in the face of unreliable networks very challenging.

[4]Most protocols make it easy to recognize any causal dependencies of packets using sequence or epoch numbers so local buffering of packets delivered prior to causal dependencies effectively implements a causality respecting network from one with eventual delivery from a clients point of view.

[5]We also note that, unlike almost all other protocols in this work, the Matrix protocol has little to no forward secrecy, though we believe this could be fixed relatively easily; albeit at the likely cost to availability in the case of failure and device loss [ACDJ23].

1.1 Our Contribution

In this work we pave the way towards scalable secure group messaging (and CGKA) over unreliable communication infrastructure. That is, we define and construct a new type of CGKA that supports processing packets in any causally respecting order. This removes the main roadblock preventing natural fork resolution for CGKA based messaging. We do so without compromising on the standard CGKA security properties. Furthermore, we provide a simple abstraction clients can use to reason and communicate about their local states, and to help determine what events (i.e. packets) they are missing and which missing ones they could still process.

Fork-resilient CGKA. We begin with a fresh approach to CGKA, called *fork-resilient CGKA* (FR-CGKA). We observe that, up till now, a typical way of thinking about regular CGKA relies heavily on the concept of a client's *current epoch.* For example, this resulted in defining authenticity for CGKA to mean that a member only accepts packets from group members in a single, i.e. the receiver's "current", epoch [ACJM20, AJM22, AHKM22a]. Consequently, all CGKA protocols to date are incapable of processing packets with events pertaining to older epochs.

To address this, a key conceptual novelty of FR-CGKA is to replace the focus on a current epochs with a focus on a current *view of the group's history.* This history is represented as a directed tree of epochs called the *history graph* [ACDT21] where an edge from an epoch E_0 to E_1 represents an event modifying E_0's state, giving rise to the new epoch E_1. Each new FR-CGKA protocol packet corresponds to creating a new edge and epoch in the history graph. Clients can create a child epoch E_1 of any epoch E_0 in their view, i.e., they can send a packet *from epoch* E_0 to convey to other group members some event changing E_0's state defining the new epoch E_1. Clients can also receive packets sent from any epoch E_0 in their view thus adding the new epoch node E_1 to their view and connecting it as a child of E_0. We adapt the standard CGKA security notions accordingly. For example, authenticity now requires that if a member accepts a packet as having been sent by some client C, then C is a member in E_0 and C did in fact create E_1. In particular, correctness of FR-CGKA ensures that packets can be processed in any causality respecting order.

Our formal model for defining FR-CGKA is based on server-aided CGKA [AHKM22a] inheriting all of its advantages such as a flexible definition parameterized by security predicates and enabling better efficiency, as members receive personalized, smaller packets.

The Pebbling Abstraction For Local States. To implement the natural fork resolution paradigm, clients that re-establish communication after a fork must determine which packets they have received (and which not) as well as which of the packets they didn't receive, yet which they are still able to process.

To help with this, we introduce a comparatively simple abstraction of a clients local state which captures precisely this (and no more) information about the client. In essence, it captures which epochs and their relations a client is aware of, for which of those epochs it can still send and receive and for which it still knows the epoch key. In a bit more detail, we call the abstraction the client's *pebbled history graph.* At any given moment, the local state of a client can be represented using this abstraction. There are three types of pebbles which can be placed on nodes (i.e. on epochs in the history graph): *move* pebbles, *visited* pebbles and *key* pebbles. A node can have at most one pebble of each type on it. A move pebble on epoch E denotes that the client can send and receive from E. A visited pebble on E denotes that the packet leading to E can *not* be processed any more. (Not being able to process previously received packets is crucial to ensure forward secrecy of messages in a CGKA-based messaging protocol.) Finally, a *key* pebble on epoch E indicates that the client still has the epoch key in their local state. Move and key pebbles can be deleted from old epochs at any time (as this simply corresponds to deleting cryptographic keys from the client's local state). A visited pebble E cannot be removed though as its presence implies that some critical cryptographic secret required to process the packet for the E's incoming edge has already been deleted. To summarize; to be able to process a packet sent from epoch E_0 to create child epoch E_1 the client must already have a move pebble on E_0 and, if E_1 is already in their view, then have no visited pebble on E_1. Processing the packet then (adds E_1 as a child of E_0 if not already there and) places a move, visited and key pebble on E_1. Thus, the pebbled history graph of a client fully determines which packets it has and still can process. We give a simple protocol in the FR-CGKA hybrid model for clients to obtain and manipulate their current pebbling state. This presents a minimal interface to the underlying FR-CGKA for use by a fork resolution procedure. In particular, it abstracts away all details about the underlying cryptographic state.

We note that FR-CGKA is a generalization of CGKA in the following sense. A CGKA can be built black-box from any FR-CGKA by keeping only the newly placed move pebble and removing all old ones whenever a new packet is processed.

Natural Fork Resolution Example for FR-CGKA. As a brief proof-of-concept demonstrating how FR-CGKA and the pebbling abstraction can be used for fork resolution we give an example natural fork resolution procedure (in Sec. 6). At a high level, when users discover that they are out of sync, they exchange their respective pebbled history graphs and missing packets and run a local state resolution algorithm to get a "current" group state consistent with their combined views. Then one of them chooses some epoch E and creates its child E_0 (or a chain of descendants ending with E_0) whose state matches the "current" one. (E should be chosen so that the cost of creating and/or transitioning to E_0 is minimized.) For state resolution, we can, for example, use the algorithm of [Fou23b], as the "event graphs" it uses are similar to history graphs.

Practical FR-CGKA protocol. As our next contribution we give a practical protocol, FREEK, based on the SAIK protocol [AHKM22a], which in turn builds on MLS [BMO$^+$18].

It turns out that modifying MLS (or SAIK) without compromising on security is not entirely straightforward. For example, the distributed MLS (dMLS) protocol of [Mat23b], which modifies MLS to allow processing packets in any causally respecting order, ends up with far weaker forward secrecy (FS) than FS-CGKA (and MLS) guarantee. Concretely, to enable processing packets from old epochs, dMLS stores old secrets (thus, removing MLS's instructions to delete them). So for example, corrupting Alice while she has a move pebble on epoch E can reveal the epoch keys (and thus messages sent by clients in those epochs since dMLS is a CGKA-based messaging protocol) of many descendant epochs of E, *even those she has already deleted.* In contrast, FREEK ensures that these keys are secure.

The main idea of FREEK is as follows. SAIK (and MLS) derive the epoch key of an epoch E_1 by mixing secret entropy from the state of its parent epoch E_0 with fresh entropy sampled by the client creating E_1. Concretely, `init_secret`, which is part of E_0's cryptographic state, is hashed with a fresh `commit_secret` distributed in a packet to the group by the creator of E_1. Recall that CGKA must provide strong FS for epoch keys. Unfortunately, the decryption keys clients use to recover `commit_secret` from a packet may not be rotated out for many epochs into the future which means we get weaker than desired FS for `commit_secret`. So instead, SAIK (and MLS) improve the FS of E_1's epoch keys by immediately deleting the `init_secret` of E_0. Thus, a leaked client's state might reveal enough to allow recovering `commit_secret` but not `init_secret` which is also needed to recompute the epoch key.

However, this creates a problem for FR-CGKA, as the deleted `init_secret` is needed to transition to other child epochs of E_0. Therefore, if the packets for those epochs arrive later, the client can no longer process them.

Based on this observation, we take a more fine-grained approach to enforcing FS by being more selective about which information is deleted. In particular, in FREEK the `init_secret` of E_0 is now the key to a puncturable PRF (PPRF). Puncturable PRFs allow puncturing their key on any subset of inputs S such that the key reveals nothing about the output of the PRF on those inputs but can still be used to evaluate the PRF on all other inputs. The epoch key of E_1 is now derived by mixing a fresh `commit_secret` with the PPRF output produced using `init_secret` of E_0 as the PPRF key and a *challenge* distributed in the packet by the creator of E_1. So, instead of deleting `init_secret`, FREEK only punctures it on the challenge, providing FS for E_1's epoch key while maintaining the ability to process future packets with other `commit_secret` values.

It remains to choose the challenge, which turns out to be a bit tricky. For example, using `commit_secret` is a bad idea, because a punctured PPRF key may reveal the inputs it was punctured on (this is actually the case for the common GGM-based PPRF). Therefore, using such a construction would leak old commit secrets, making FS worse again. Using instead a random challenge (attached to the packet) is also a bad idea, because it opens the door to denial-of-service attacks (and it increases bandwidth). In such an attack, a malicious group member Malory creates a packet P_1 with the challenge copied from some honest packet P_2, and delivers P_1 to Bob before P_2. As a result, Malory blocks Bob from receiving P_1, since Bob already punctured his `init_secret`. As a result, FREEK constructs the challenge from a cryptographic commitment to the new epoch's public state and half of the commit secret (the other half becomes what was previously the commit secret). See Sec. 4 for more details on challenge selection.

Optimally secure FR-CGKA. Finally, we construct a second FR-CGKA protocol with optimal security predicates. That is, all epochs whose security does not contradict the protocol's correctness are secure. To achieve this, we expand the techniques of [ACJM20] which constructs standard CGKA with optimal security using hierarchical identity-based encryption (HIBE). To extend the CGKA of [ACJM20] to the FR-CGKA setting, parties would have to store old CGKA states, which destroys optimal security. We show how to fix this using HIBE with a binary identity space, which achieves both HIBE and puncturable PKE[GM15].

1.2 Related Work

Besides those already mentioned in the introduction, there various works about fork resolution and about CGKA. First, several (also non-cryptographic) decentralized systems in practice can be cast as using resolution algorithms to provide consistency, some more explicitly than others. For example some messaging applications [Fou23a, CEST22, OR93, Jab23], most blockchain protocols (e.g. [Nak08, But14]) and CRDT based systems like collaborative document editing applications [Goo23, Aut23] all use either implicit or even explicit [Fou23a] resolution algorithms. Further, [DDF21] provides a way to deal with a different type of fork; namely ones caused by malicious insiders deviating from the honest protocol in the setting with a server trusted to deliver packets in order.

Another related line of work considers *concurrent CGKA* protocols [AAN$^+$22b, AAN$^+$22a, BDG$^+$22, BMO$^+$18] which allow for concurrent PCS updates (e.g. due to a network partition). However for these protocols, concurrency is only supported within one epoch. That is, once a client decides to apply one or more concurrently generated PCS updates (or other modifications) to its local state, it enters a new epoch at which point it can no longer process any other incoming events pertaining to the previous epoch. In particular, it has no way of processing delayed packets pertaining to the old group state. As such, these protocols only provide a very limited type of fork resilience.

The most prominent CGKA protocol, the one underlying the MLS[BMO$^+$18] messaging standard, received a lot of attention from academia. The works [BCK21, AJM22] provide cryptographic analysis of the whole protocol, while [BCK22] analyzes its key schedule. Further, [WPBB22] gives a formally verified implementation of MLS with security proofs in F*. Apart from MLS, other CGKA protocols have been proposed, improving efficiency and/or security. In terms of security, RTreeKEM [ACDT20] improves PCFS while [HKP22] provides better metadata privacy. The works of [BCV22, KEO+22] consider CGKA with flexible authorization. Protocols with better efficiency (at least in certain scenarios) include TTKEM [ACC+21] and the protocol of [AAB+21]. The work [BDT22] proposes a (potentially) more efficient but also more restricted variant of CGKA. Another way to improve efficiency is to consider *server-aided* CGKA [KKPP20, HKP+21, AHKM22a], where parties download (smaller) personalized packets prepared by an untrusted mailboxing service (instead of communicating via broadcast as is typical for CGKA).

Finally, for an overview of game-based security models for group key exchange see [**?**].

2 Preliminaries

2.1 Notation

For a vector $\vec{v}$ and a value v, we denote by $\vec{v} \parallel v$ appending v at the end of $\vec{v}$. The empty vector is denoted ϵ.

2.2 Binary-Tree Encryption and HIBE

Hierarchical identity-based encryption (HIBE) [GS02] is public-key encryption where secret keys form a hierarchy (the public keys form the analogous hierarchy). On top is the master secret key. Directly below it are keys associated with identities I from some identity space, w.l.o.g. $\{0,1\}^\ell$ for some ℓ. Directly below each key with identity I_1 are keys associated with two-element identity vectors (I_1, I_2) for $I_2 \in \{0,1\}^\ell$, and so on. This hierarchy can be visualized as a tree where each node is associated with an identity vector $\vec{I}$ and the corresponding secret key. On top is the root with the empty identity vector $\vec{I} = \epsilon$ and the master secret key. Then nodes on depth d have identity vectors of length d.

Given the master public key, one can encrypt a message to any public key in the hierarchy. Further, given a secret key, one can derive all secret keys below it in the hierarchy. However, the secret key reveals no information about messages encrypted to public keys that are not below its corresponding public key.

Binary-Tree Public-Key Encryption (BtPke) is a special case of HIBE where the identities I are bits, i.e. $\ell = 1$. Our protocol only uses BtPke, so we only define the special case. The syntax is as follows.

Key Generation: $\mathsf{BtPke.gen}() \to (\mathsf{epk}, \mathsf{esk})$ generates a fresh master public and secret key.

Sub-key Derivation: $\mathsf{BtPke.der}(\mathsf{esk}) \to (\mathsf{esk}_0, \mathsf{esk}_1)$ takes as input a secret key for an (implicit) binary identity vector $\vec{I}$ and outputs keys for identity vectors $\vec{I} \parallel 0$ and $\vec{I} \parallel 1$, respectively.

Encryption: $\mathsf{BtPke.enc}(\mathsf{epk}, \vec{I}, m) \to c$ takes as input a master public key, a binary identity vector $\vec{I}$ and a message m, and outputs a ciphertext c.

Decryption: $\mathsf{BtPke.dec}(\mathsf{esk}, c) \to m/\perp$ takes the public a secret key esk for an (implicit) binary identity $\vec{I}$ and a ciphertext c encrypted to $\vec{I}$ and returns either a decrypted message m or $\perp$ if decryption fails.

Game IND-CCA

```
$\underline{\mathrm{Exp}^{\textsf{IND-CCA}}_{\textsf{BtPke},b}(\mathcal{A})}$
    $(\mathsf{epk}, \mathsf{esk}) \leftarrow \mathsf{BtPke.gen}()$
    $\mathsf{C} \leftarrow \varnothing$
    $c^*, \vec{I}^* \leftarrow \bot$
    $(m_0, m_1, \vec{I}^*) \leftarrow \mathcal{A}^{\mathrm{Corr},\mathrm{Dec}}(\mathsf{epk})$
    $c^* \leftarrow \mathsf{BtPke.enc}(\mathsf{epk}, \vec{I}^*, m_b)$
    $b' \leftarrow \mathcal{A}^{\mathrm{Corr},\mathrm{Dec}}(c^*)$
    req $\nexists \vec{I} \in \mathsf{C}, \vec{I}' \in \{0,1\}^* : \vec{I}^* = \vec{I} \parallel \vec{I}'$
    return $b = b'$
```

```
Oracle Corr($\vec{I}$)
    $\mathsf{C} \mathrel{+}\!\leftarrow \vec{I}$
    return *get-esk($\mathsf{esk}, \vec{I}$)

Oracle Dec($\vec{I}, c$)
    if $c^* \neq \bot$ then
        req $(\vec{I}, c) \neq (\vec{I}^*, c^*)$
    $\mathsf{esk}' \leftarrow$ *get-esk($\mathsf{esk}, \vec{I}$)
    return $\mathsf{BtPke.dec}(\mathsf{esk}', c)$
```

```
Helper *get-esk($\mathsf{esk}, \vec{I}$)
    $\mathsf{esk}' \leftarrow \mathsf{esk}$
    for $i = 0$ to $|\vec{I}|$ do
        $(\mathsf{esk}'_0, \mathsf{esk}'_1) \leftarrow \mathsf{BtPke.der}(\mathsf{esk}')$
        $\mathsf{esk}' \leftarrow \mathsf{esk}'_{\vec{I}[i]}$
    return $\mathsf{esk}'$
```

Figure 1: Experiment defining security of BtPke.

Correctness requires that for any $(\mathsf{epk}, \mathsf{esk})$ in the support of $\mathsf{BtPke.gen}$, for any $d \geq 0$ and $\vec{I} \in \{0,1\}^d$, for any $\mathsf{esk}_{\vec{I}}$ generated by running $\mathsf{BtPke.der}$ with all bits of $\vec{I}$, and for any m, we have $\mathsf{BtPke.dec}(\mathsf{esk}_{\vec{I}}, \mathsf{BtPke.enc}(\mathsf{epk}, \vec{I}, m)) = m$ with probability 1.

For security, we require the standard, straightforward modification of IND-CCA security. Formally,

Definition 1. *The advantage of an adversary $\mathcal{A}$ against* IND-CCA *security of a scheme* BtPke *is defined as*

$$\mathrm{Adv}^{\textsf{IND-CCA}}_{\textsf{BtPke}}(\mathcal{A}) = \Pr[\mathrm{Exp}^{\textsf{IND-CCA}}_{\textsf{BtPke},1}(\mathcal{A}) \Rightarrow 1] - \Pr[\mathrm{Exp}^{\textsf{IND-CCA}}_{\textsf{BtPke},0}(\mathcal{A}) \Rightarrow 1],$$

where the experiment $\mathrm{Exp}^{\textsf{IND-CCA}}_{\textsf{BtPke},b}(\mathcal{A})$ is defined in Fig. 1.

2.3 Binary-Tree Signatures

Binary-tree signatures, BtSig, are analogous to binary-tree encryption. Secret and public keys form a hierarchy. A master public key allows to verify signatures for all binary identity vectors $\vec{I}$. A secret key for a vector $\vec{I}$ allows to generate signatures for $\vec{I}$ and derive keys for identity vectors $\vec{I} \parallel 0$ and $\vec{I} \parallel 1$.

BtSig is a special case of hierarchical identity-based signatures [GS02].

Key Generation: $\mathsf{BtSig.gen}() \to (\mathsf{spk}, \mathsf{ssk})$ generates a fresh master public key and master secret key for identity ϵ.

Sub-key Derivation: $\mathsf{BtSig.der}(\mathsf{ssk}, I) \to \mathsf{ssk}$ takes as input a secret key for an (implicit) binary identity vector $\vec{I}$ and secret keys for identity vectors $\vec{I} \parallel 0$ and $\vec{I} \parallel 1$, respectively.

Signing: The signing algorithm $\mathsf{BtSig.sig}(\mathsf{ssk}, m) \to \sigma$ takes as input a secret key for an (implicit) binary identity vector $\vec{I}$ and a message m, and outputs a signature σ over m for $\vec{I}$.

Verification: The verification algorithm $\mathsf{BtSig.ver}(\mathsf{spk}, \vec{I}, m, \sigma) \to b$ takes as input a (master) public key, a binary identity vector $\vec{I}$, a message m and a signature σ, and outputs $b = \texttt{true}$ if σ is a valid signature over m for $\vec{I}$.

Correctness requires that for any $(\mathsf{spk}, \mathsf{ssk})$ in the support of $\mathsf{BtSig.gen}$, for any $d \geq 0$ and $\vec{I} \in \{0,1\}^d$, for any $\mathsf{ssk}_{\vec{I}}$ generated by running $\mathsf{BtSig.der}$ with all bits of $\vec{I}$, and for any m, we have $\mathsf{BtSig.ver}(\mathsf{spk}, \vec{I}, \mathsf{BtSig.sig}(\mathsf{ssk}_{\vec{I}}, m), m) = 1$ with probability 1. Security is formalized as follows.

Game $\mathsf{EUF\text{-}CMA}_{\mathsf{BtSig},\mathcal{A}}$

```
$\mathrm{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{BtSig.sig}}(\mathcal{A})$
    $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \mathsf{BtSig.gen}()$
    $\mathsf{C}, \mathsf{S} \leftarrow \varnothing$
    $(\sigma^*, m^*, \vec{I}^*) \leftarrow \mathcal{A}^{\mathrm{Corr},\mathrm{Sig},\mathrm{Upd}}(\mathsf{spk})$
    $\mathsf{win} \leftarrow 1$
    $\mathsf{win} \leftarrow \mathsf{win} \wedge \mathsf{BtSig.ver}(\mathsf{spk}, \vec{I}^*, m^*, \sigma^*)$
    $\mathsf{win} \leftarrow \mathsf{win} \wedge (m^*, \vec{I}^*) \notin \mathsf{S}$
    $\mathsf{win} \leftarrow \mathsf{win} \wedge \nexists \vec{I} \in \mathsf{C}, \vec{I}' \in \{0,1\}^* : \vec{I}^* = \vec{I} \,\|\, \vec{I}'$
    return $\mathsf{win}$
```

```
Oracle Sig($\vec{I}, m$)
    $\mathsf{ssk}' \leftarrow$ *get-ssk($\mathsf{ssk}, \vec{I}$)
    $\sigma \leftarrow \mathsf{BtSig.sig}(\mathsf{ssk}', m)$
    $\mathsf{S} \mathrel{+}\!\!\leftarrow (m, \vec{I})$
    return $\sigma$
Oracle Corr($\vec{I}$)
    $\mathsf{C} \mathrel{+}\!\!\leftarrow \vec{I}$
    return *get-ssk($\mathsf{ssk}, \vec{I}$)
```

```
Helper *get-ssk($\mathsf{ssk}, \vec{I}$)
    $\mathsf{ssk}' \leftarrow \mathsf{ssk}$
    for $i = 0$ to $|\vec{I}|$ do
        $(\mathsf{ssk}'_0, \mathsf{ssk}'_1) \leftarrow \mathsf{BtSig.der}(\mathsf{ssk}')$
        $\mathsf{ssk}' \leftarrow \mathsf{ssk}'_{\vec{I}[i]}$
    return $\mathsf{ssk}'$
```

Figure 2: The EUF-CMA game for key-updatable signatures.

Definition 2. *The advantage of an adversary $\mathcal{A}$ against* EUF-CMA *security of a scheme* BtSig *is defined as*

$$\mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{BtSig}}(\mathcal{A}) = \Pr[\mathrm{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{BtSig}}(\mathcal{A}) \Rightarrow 1],$$

where the experiment $\mathrm{Exp}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{BtSig}}(\mathcal{A})$ *is defined in Fig. 2.*

2.4 Collision-Resistant PRF

Let $\mathsf{CR\text{-}PRF}(k, x) \to y$ be a function. We require two security properties for CR-PRF: standard PRF security and collision-resistance (CR) w.r.t. (k, x) pairs. That is, CR requires that no efficient adversary can find two pairs (k, x) and (k', x') such that $\mathsf{CR\text{-}PRF}(k, x) = \mathsf{CR\text{-}PRF}(k', x')$. CR is not implied by PRF security because the former implies there are no collision on *keys* too.

2.5 Puncturable PRF

A puncturable pseudorandom function [BW13, KPTZ13, BGI14], PPRF, is a variant of a PRF that additionally allows to puncture the secret key on different inputs. A key punctured on an input x reveals no information about the output on x.

For simplicity, we consider PPRFs with key, input and output space of bitstrings of length κ (the security parameter). Formally, a PPRF consists of the following algorithms

Evaluation: $\mathsf{PPRF.eval}(k, x) \to y$ takes as input a key $k \in \{0,1\}^\kappa$ (chosen uniform at random) and an input $x \in \{0,1\}^\kappa$ and outputs $y \in \{0,1\}^\kappa \cup \{\bot\}$

Puncturing: $\mathsf{PPRF.puncture}(k, x) \to k'$ takes as input a key $k \in \{0,1\}^\kappa$ and an input $x \in \{0,1\}^\kappa$ and outputs a key $k' \in \{0,1\}^\kappa$ punctured on x

Correctness. A PPRF is correct if for any $n > 0$, for any $k, x_1, \ldots, x_{n-1} \in \{0,1\}^\kappa$ and $x \in \{0,1\}^\kappa \setminus \{x_1, \ldots, x_n\}$, if we set $k_1 = k$ and $k_{i+1} = \mathsf{PPRF.puncture}(k_i, x_i)$ for $i \in [1, n-1]$, then $\mathsf{PPRF.eval}(k_n, x) = \mathsf{PPRF}(k, x) \neq \bot$. Note that correctness implies commutativity.

We note that typically a weaker PPRF correctness is considered, where a key is only punctured once. We need the stronger version. It can be easily achieved.

Game OW-PPRF

```
$\mathrm{Exp}^{\mathsf{OW\text{-}PPRF}}_{\mathsf{PPRF},N,n}(\mathcal{A})$
    for $i \in [N]$ do
        $k_i \xleftarrow{\$} \{0,1\}^\kappa$
    for $(i,j) \in [N] \times [n]$ do
        $r_{i,j} \xleftarrow{\$} \{0,1\}^\kappa$
    $\mathsf{C} \leftarrow \varnothing$
    $\mathsf{win} \leftarrow \mathtt{false}$
    $\mathcal{A}^{\mathrm{Corr},\mathrm{Eval}}(r_{i,j} \mid (i,j) \in [N] \times [n])$
    return win

Oracle Corr$(i, j_1, \ldots, j_\ell)$
    req $i \in [N] \wedge j_1, \ldots, j_\ell \in [n]$
    $k \leftarrow k_i$
    for $j \in j_1, \ldots, j_\ell$ do
        $k \leftarrow \mathsf{PPRF.puncture}(k, r_{i,j})$
    $\mathsf{C} \mathrel{+}\leftarrow ([N] \times [n]) \setminus \{(i,j_1), \ldots, (i,j_\ell)\}$
    return $k$

Oracle Try$(i,j,y)$
    req $(i,j) \in [N] \times [n]$
    $y' \leftarrow \mathsf{PPRF.eval}(k_i, r_{i,j})$
    if $(i,j) \notin \mathsf{C} \wedge y = y'$ then
        $\mathsf{win} = \mathtt{true}$
    return $y = y'$

Oracle Eval$(i,x)$
    req $i \in [N] \wedge \nexists j : x = r_{i,j}$
    return $\mathsf{PPRF.eval}(k_i, x)$
```

Figure 3: Experiment defining security of PPRF schemes.

Security. We define a non-standard security notion for PPRF called one-wayness (with adaptive corruptions), OW-PPRF. The game is defined in Fig. 3. Roughly, the challenger samples N keys and for each key n random inputs. The adversary's goal is to compute one of these inputs given the outputs (hence, one-wayness). In addition, the adversary can use a corrupt oracle which outputs a challenge key punctured on a combination of challenge inputs chosen by the adversary. Such adaptive corruptions are necessary to prove security of our protocol FREEK. We are not aware of any work that defines such a notion.

Definition 3. *Let N and n be positive integers. The advantage of an adversary $\mathcal{A}$ against* OW-PPRF *security of a scheme* PPRF *is defined as*

$$\mathrm{Adv}^{\mathsf{OW\text{-}PPRF}}_{\mathsf{PPRF},N,n}(\mathcal{A}) = \Pr[\mathrm{Exp}^{\mathsf{OW\text{-}PPRF}}_{\mathsf{PPRF},N,n}(\mathcal{A}) \Rightarrow 1],$$

where the experiment $\mathrm{Exp}^{\mathsf{OW\text{-}PPRF}}_{\mathsf{PPRF},N,n}(\mathcal{A})$ is defined in Fig. 3.

Construction. As shown in [BW13, KPTZ13, BGI14], the PRF by Goldreich et al. [GGM84] (GGM) naturally extends to a puncturable PRF.

Roughly, the GGM construction defines a binary tree of height κ where each node is labeled by a κ-bit value. Nodes are identified by binary strings: root is identified by an empty string ϵ, and if a node is identified by a string x, then its left and right children are identified by $x \parallel 0$ and $x \parallel 1$, respectively. To define the labels, the scheme uses a length-doubling pseudorandom generator PRG. The root's label is the secret PRF key. For a node with label ℓ, the labels of its children are the first κ bits and the last κ bits of $\mathsf{PRG}(\ell)$, respectively. The output on input x is the label of the node identified by x.

To get a PPRF, we need to implement puncturing. At a high level, to puncture on x, we store the labels of all nodes on the copath from the node identified by x to the root, i.e., those with identifiers $(x_1, x_2, ..., \neg x_n)$ for some $n \in [0, \kappa]$ where $(x_1, x_2, ..., x_n)$ is a prefix of x and $\neg$ is binary negation. We delete the root's label. Note that we can compute the labels of all leaves except x.

OW-PPRF **security of the construction.** We note that even for standard PPRF security (with one adaptive corruption of a key punctured on one challenge value), there is no polynomial-time reduction to PRG security for the GGM PPRF. However, it is easy to see that the construction is OW-PPRF secure if PRG is modeled as a random oracle.

Since we only require one-wayness, there may be a standard-model proof showing a reduction from OW-PPRF security of GGM to some other property of PRG. Since we use the construction in

FREEK whose proof is in the ROM anyway, this would not improve our result in particular. It is an interesting question, however.

3 Fork-Resilient CGKA

In the current section we define the notion of Fork-Resilient Continuous Group Key Agreement (FR-CGKA). Our model is based on the model of [AHKM22a], which in turn builds on [ACJM20], i.e., our definitions and protocols are in the server-aided CGKA setting. In Sec. 3.1 we first recall the core ideas behind CGKA in general, and then, in Secs. 3.2 and 3.3, we present our FR-CGKA definition.

3.1 (Server-aided) CGKA

A CGKA protocol enables a dynamic group of parties to continuously agree on shared, symmetric secret keys, while ensuring the *confidentiality* and *authenticity* of those keys. The protocol is run over a network and parties may be online/offline at arbitrary times, while the communication between parties is assumed to be handled by an untrusted mailboxing service. A PKI is also assumed, that enables parties to exchange initial key material so that they can be added while offline.

Syntax. A CGKA protocol execution proceeds in *epochs*. Each epoch defines a fixed set of group properties, most importantly the member set and the shared group key. Any group member can modify the group properties, which means that they create a new epoch. In this work, we consider three types of operations they can do: (1) the *addition* of new members, (2) the *removal* of existing ones, and (3) the *update* of the group key. Each protocol operation is performed by creating a single message, which is uploaded to the untrusted mailboxing service. Afterwards, each group member can download a possibly personalized message, and, if they accept it, they transition to the new epoch. When adding a member, a CGKA protocol may contact a *key service* to fetch a so-called key package previously uploaded by the added member. This way the added member does not need to be online. A key package can only be used once and it must be group-agnostic — a member does not know to which groups they will be added.

Adversary. The adversary can fully control the mailboxing service and repeatedly expose secret states for parties. Note that this combination allows it to inject messages on behalf of parties whose states were exposed (they have no more secrets). Intuitively, this means that if Alice's state is exposed and her message is delivered first, then she can heal. However, if a message injected on her behalf is delivered first, then Alice is doomed — the only way for the group to heal is to remove the adversary impersonating Alice and add Alice again.

History graphs. A useful tool when thinking of CGKA are *history graphs* introduced in [ACDT21]. Intuitively, a history graph is a symbolic representation of group evolution. Epochs are represented as nodes and group modifications are represented as directed edges. For instance, assume a group with a single member, Alice in epoch U. If Alice decides to add Bob, she creates an epoch V with an edge from U to V, and after Bob joins they are both in epoch V. If Alice then decides to update the group key, she creates an epoch W (every epoch has its own key). The history graph also stores information about parties' current epochs, adversary's actions, etc.

Ideally, the history graph of a CGKA protocol execution should be a chain, i.e., each epoch should have only a single child epoch in which the epoch creator transitions to it instantly, while other parties transition as soon as they go online (or they join the group) and process the creator's message.

However, this is not true since forks can also be created: if two parties simultaneously create new epochs, an active adversary can deliver different messages to different parties, causing them to follow different branches, creating a fork. Furthermore, it can make parties join fake groups by injecting messages that invite them. Epochs in fake groups form what we call *detached trees*, where in presence of such trees the history graph becomes a *forest*.

Security. Using the language of history graphs, [AHKM22a] identifies the following CGKA security properties. The first two properties generalize PCS and FS. We also add correctness which is not considered in [AHKM22a].

Agreement: Any two parties in the same epoch agree on the group state, i.e., the set of current members, the group key, the last group modification and the previous epoch. One consequence of the property is agreement on the transcript, that is, any two parties in a given epoch reached it by executing the same sequence of group modifications since the latter one joined.

Confidentiality: An epoch U is confidential if the adversary has no information about its group key. An active adversary may destroy confidentiality in certain epochs. CGKA security is parameterized by a *confidentiality predicate* which decides if an epoch U is confidential.

Authenticity: Authenticity for a party A in an epoch U is preserved if the following holds: If a party in U transitions to a child epoch V and identifies A as the sender creating V, then A indeed created V. Again, an active adversary may destroy authenticity for certain epochs and parties. CGKA security is parameterized by an *authenticity predicate* which decides if authenticity of a party A in epoch E is preserved.

Strong Correctness: An honestly-generated message transitioning from epoch U to V is accepted by a party in U. This holds even in the presence of corruptions, e.g. when U is created by the adversary (via injections).

3.2 FR-CGKA Protocols

The intuition from the previous subsection leads to CGKA protocols and definitions which make fork resolution very problematic. In such CGKAs, each party knows only about *one current* epoch. If two parties end up with current epochs in different branches, they will never be able to reach the same current epoch and, hence, communicate; see the example in Fig. 4. The reason is that agreement requires a common group history (including all ancestors of the current epoch), and reaching the same epoch from different branches clearly contradicts that.

FR-CGKA enables fork resolution without giving up agreement. At a high level, we replace the notion of a party's current epoch by *its current view of the history graph.* This means that a party is not in one epoch but in many epochs at once. In the FR-CGKA syntax, when a party creates a new epoch, it specifies a history-graph node which should be its parent. When a party receives a message, it simply learns about a new node (and its parent). We say that it traverses the edge in such case. Further, it can fetch the group keys from different nodes.

More precisely, a party's view is represented by a pebbling of the history graph. See the example in Figs. 5 to 7. Each party has three types of pebbles: A party A has a *move pebble* on an epoch U if it has information required to transition to (non-visited) children of U. That is, A can create a child epoch of U and transition to a child of U created by another party. A's *key pebble* on U indicates that A can output the group key for epoch U. A key can only be fetched once for forward secrecy. Finally, forward secrecy requires that after traversing an edge from U to V, a party A removes the information needed to do that. A's *visited pebble* on V denotes that A already traversed the edge.

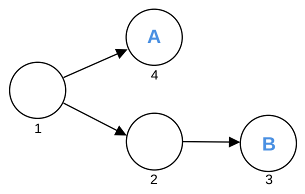


Figure 4: A fork blocks CGKA : Alice (A) in epoch 4 will never be able to read messages sent by Bob (B) in epoch 3.

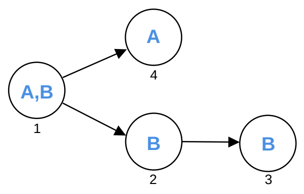


Figure 5: Resolving a fork in FR-CGKA : Alice and Bob can still process messages from epoch 1, i.e., each has a move pebble on epoch 1. In addition, Bob has move pebbles on epochs 2 and 3, and Alice has a move pebble on epoch 4.

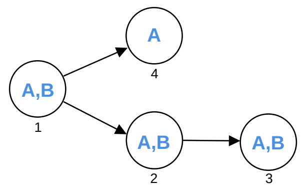


Figure 6: Resolving a fork in FR-CGKA : Once Alice receives messages $(1, 2)$ and $(2, 3)$, she gets move pebbles on epochs 2 and 3 and she can talk to Bob (she needs *both* messages to get to epoch 3.)

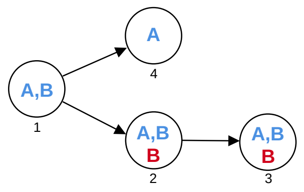


Figure 7: Forward secrecy of FR-CGKA : Bob's state does not contain information needed to transition to epochs he already visited, i.e., 2 and 3, marked by his visited pebbles. This means that if Bob is corrupted in the situation above, the adversary learns the group key in epoch 4 (this is possible for any correct FR-CGKA) but not in 2, 3.

In addition, the syntax of FR-CGKA allows for deleting move pebbles. It provides FS for old epochs from which a party no longer needs to move.

All security properties of CGKA also apply to FR-CGKA. The only difference is that agreement now mandates that for any epoch *in the view of* two parties, the parties agree on the group state in this epoch. Note that when the state of a party A is exposed, the adversary gets the ability to traverse edges from all nodes on which A has a move pebble except those on the ends of which she has a visited pebble.

3.3 FR-CGKA Security Definition

We define security of FR-CGKA protocols in the UC framework, i.e., we consider a real-world experiment where an environment $\mathcal{A}$ interacts with an FR-CGKA protocol π, denoted by $\text{REAL}_\pi(\mathcal{A})$, and an ideal-world experiment where $\mathcal{A}$ interacts with an ideal FR-CGKA functionality $\mathcal{F}_{\text{FR-CGKA}}$ and a simulator $\mathcal{S}$, denoted by $\text{IDEAL}_{\mathcal{F}_{\text{FR-CGKA}},\mathcal{S}}(\mathcal{A})$. In this setting, a protocol π is secure if for all $\mathcal{A}$ there exists an $\mathcal{S}$ such that the difference between the probability that $\mathcal{A}$ outputs 1 in $\text{REAL}_\pi(\mathcal{A})$ and the probability that $\mathcal{A}$ outputs 1 in $\text{IDEAL}_{\mathcal{F}_{\text{FR-CGKA}},\mathcal{S}}(\mathcal{A})$, is negligible. Due to space limitations, basic ideas behind the UC framework and the security model of [AHKM22a] are deferred to App. A.

We now formally define our ideal functionality, $\mathcal{F}_{\text{FR-CGKA}}$, presented in Fig. 8. The functionality is parameterized by the confidentiality and authenticity predicates, **conf** and **auth**, respectively, the packet extraction function $\texttt{Ext}(C, \mathsf{id}) \to c$ and the id of the group creator, denoted by $\mathsf{id}_{\mathsf{creator}}$. Note that our security definition is generic enough to capture arbitrary predicates **conf**, **auth**, admitting protocols with different levels of security. The (deterministic) $\texttt{Ext}$ function is (only) used to express correctness. $\texttt{Ext}$ is defined by the FR-CGKA protocol realizing $\mathcal{F}_{\text{FR-CGKA}}$ for the mailboxing service to extract from an uploaded packet C, a packet c downloaded by id.

Functionality $\mathcal{F}_{\text{FR-CGKA}}$

Parameters: **conf**, **auth**, packet extraction function $\texttt{Ext}(C) \to c$, group creator $\mathsf{id}_{\mathsf{creator}}$

```
Initialization // Executed on first input.
  $\mathsf{Eps}[*] \leftarrow \bot$
  Receive $U_0$ from the simulator
  $\mathsf{Eps}[U_0] \leftarrow$ *new-ep(sndr $= \mathsf{id}_{\mathsf{creator}}$, par $= \bot$, act
                $=$ 'create', mem $= \{\mathsf{id}_{\mathsf{creator}}\}$, packet $=$ 'create')
  $\mathsf{Move}^{\mathcal{A}}, \mathsf{Key}^{\mathcal{A}}, \mathsf{Visited}^{\mathcal{A}} \leftarrow \varnothing$
  $\mathsf{Move}, \mathsf{Key}, \mathsf{Visited} \leftarrow \{(\mathsf{id}_{\mathsf{creator}}, U_0)\}$
  return $U_0$ to $\mathsf{id}_{\mathsf{creator}}$

Input (Send, $U$, act $\in \{$'up', 'add'-$\mathsf{id}_t$, 'rem'-$\mathsf{id}_t\}$) from id
  Send (Send, id, $U$, act) to sim., receive $(V, C, ack)$
  req $ack \wedge \mathsf{Eps}[V] = \bot$
  // Compute the new epoch created by the action.
  $\mathsf{Eps}[V] \leftarrow$ *new-ep(sndr $=$ id, par $= U$, act,
                          mem $=$ *mem($U$, act), packet $= C$)
  $\mathsf{Move}, \mathsf{Key}, \mathsf{Visited} \mathrel{+}\leftarrow (\mathsf{id}, V)$
  *assert-agree-auth-preserved
  return $(V, C)$

Input (Receive, $c$) from id
  Send (Receive, id, $c$) to sim.
  if $\exists V$ : $c = \texttt{Ext}(\mathsf{Eps}[V].\mathsf{packet})$ then
    req *step-correct(Move, Key, Visited, id, $V$)
  else
    Receive $(ack, V)$ from the sim.; req $ack$
  if $\mathsf{Eps}[V] \neq \bot$ then
    (sndr, act, $U$) $\leftarrow$ ($\mathsf{Eps}[V]$.sndr, $\mathsf{Eps}[V]$.act, $\mathsf{Eps}[V]$.par)
  else
    Receive (sndr, act, $U$) from sim.
    $\mathsf{Eps}[V] \leftarrow$ *new-ep(sndr, par $= U$, act,
                        mem $=$ *mem($U$, act), packet $=$ 'inj')
  if act $\neq$ 'rem'-id then
    $\mathsf{Move}, \mathsf{Key}, \mathsf{Visited} \mathrel{+}\leftarrow (\mathsf{id}, V)$
  *assert-agree-auth-preserved
  return $(U, V, \mathsf{sndr}, \mathsf{act})$

Input (Join, $c$) from id // Note : id may already be in the group in another epoch
  Send (Join, id, $c$) to sim., receive $(ack, U)$
  req $ack$
  if $\mathsf{Eps}[U] \neq \bot$ then
    (sndr, act, mem) $\leftarrow$ ($\mathsf{Eps}[U]$.sndr, $\mathsf{Eps}[U]$.act, $\mathsf{Eps}[U]$.mem)
  else
    Receive (sndr, act, mem) from sim.
    $\mathsf{Eps}[U] \leftarrow$ *new-ep(sndr, par $= \bot$, act, mem, packet $=$ 'inj')
  assert act $=$ 'add'-id
  $\mathsf{Move}, \mathsf{Key}, \mathsf{Visited} \mathrel{+}\leftarrow (\mathsf{id}, U)$
  *assert-agree-auth-preserved
  return $(U, \mathsf{sndr}, \mathsf{mem})$

Input (GetKey, $U$) from id
  req $(\mathsf{id}, U) \in \mathsf{Key}$
  Send (Key, id, $U$) to the sim. and receive $I$.
  if $\mathsf{Eps}[U].\mathsf{key} = \bot$ then
    if conf($U$, Eps, $\mathsf{Move}^{\mathcal{A}}, \mathsf{Key}^{\mathcal{A}}, \mathsf{Visited}^{\mathcal{A}}$) then
      $\mathsf{Eps}[U].\mathsf{key} \xleftarrow{\$} \{0,1\}^{\kappa}$; $\mathsf{Eps}[U].\mathsf{chall} \leftarrow$ true
    else
      $\mathsf{Eps}[U].\mathsf{key} \leftarrow I$
  $\mathsf{Key} \mathrel{-}\leftarrow (\mathsf{id}, U)$
  return $\mathsf{Eps}[U]$.key

Input (DeleteMovePebble, $U$) from id
  Send (DeleteMovePebble, id, $U$) to sim.
  $\mathsf{Move} \mathrel{-}\leftarrow (\mathsf{id}, U)$

Corruption (Expose, id)
  $\mathsf{Key}^{\mathcal{A}} \mathrel{+}\leftarrow \{U$ : $(\mathsf{id}, U) \in \mathsf{Key}\}$
  for $U$ s.t. $(\mathsf{id}, U) \in \mathsf{Move} \setminus \mathsf{Move}^{\mathcal{A}}$ do
    $\mathsf{Move}^{\mathcal{A}} \mathrel{+}\leftarrow (\mathsf{id}, U)$
    $\mathsf{Visited}^{\mathcal{A}} \mathrel{+}\leftarrow \{(\mathsf{id}, V)$ : $U = \mathsf{Eps}[V].\mathsf{par} \wedge (\mathsf{id}, V) \in \mathsf{Visited}\}$
  *assert-agree-auth-preserved
  // Avoids commitment problem.
  only allowed if $\nexists U : \mathsf{Eps}[U].\mathsf{chall} \wedge$
                 $\neg$conf($U$, Eps, $\mathsf{Move}^{\mathcal{A}}, \mathsf{Key}^{\mathcal{A}}, \mathsf{Visited}^{\mathcal{A}}$)
```

```
Helper *new-ep(sndr, par, act, mem, packet)
  return new epoch with given sndr, par, act,
  mem and packet, and with key $= \bot$, exp $= \varnothing$,
  chall $=$ false.

Helper *mem($U$, act)
  $G \leftarrow \mathsf{Eps}[U]$.mem
  if act $=$ 'add'-$\mathsf{id}_t$ then $G \mathrel{+}\leftarrow \mathsf{id}_t$
  else if act $=$ 'rem'-$\mathsf{id}_t$ then $G \mathrel{-}\leftarrow \mathsf{id}_t$
  req act $=$ 'up' $\vee\ G \neq \mathsf{Eps}[U]$.mem
  return $G$

Helper *assert-agree-auth-preserved
  assert HG has no cycles
  for $U$ s.t. $\mathsf{Eps}[U] \neq \bot$ do
    assert $\{\mathsf{id} : (\mathsf{id}, U) \in \mathsf{Visited}\} \subseteq \mathsf{Eps}[U].\mathsf{mem}$
    assert $\mathsf{Eps}[U].\mathsf{par} = \bot \vee \mathsf{Eps}[U].\mathsf{mem} =$ *mem($\mathsf{Eps}[U]$.par, $\mathsf{Eps}[U]$.act)
  assert $\nexists V : \mathsf{Eps}[V].\mathsf{packet} =$ 'inj' $\wedge$ auth($V$, Eps, $\mathsf{Move}^{\mathcal{A}}, \mathsf{Key}^{\mathcal{A}}, \mathsf{Visited}^{\mathcal{A}}$)

Helper *step-correct(Move, Visited, id, $V$)
  return $(\mathsf{id}, \mathsf{Eps}[V].\mathsf{par}) \in \mathsf{Move} \wedge (\mathsf{id}, V) \notin \mathsf{Visited}$
```

Figure 8: The ideal CGKA functionality.

Notation. We use the keyword **assert** followed by a condition **cond**, to restrict the simulator's actions as follows: if the condition **cond** is false, then the functionality permanently halts, making the real and ideal worlds easily distinguishable. We use **only allowed if** followed by a condition **cond** to restrict the environment. That is, our statements quantify only over environments who, when interacting with $\mathcal{F}_{\text{FR-CGKA}}$ and any simulator, never make **cond** false. We write *"Receive x from the simulator"* to denote that the functionality sends a dummy value to it, waits until it sends a value x back and asserts via **assert** that the received value is of the correct format.

History graph. The functionality $\mathcal{F}_{\text{FR-CGKA}}$ maintains a history graph represented as an array Eps, where $\mathsf{Eps}[U]$ denotes the epoch identified by U. Each epoch, say E, has a number of attributes, listed below, and we use the standard object-oriented notation to access their values, e.g., $E.\mathsf{mem}$ returns the set of group members in E. Epoch identifiers U are arbitrary, i.e., chosen by the simulator, subject to some natural conditions such as that, when a group member creates a new epoch, its identifier must not already exist in the graph.

$E.\mathsf{par}$	The parent epoch of E.
$E.\mathsf{sndr}$	The party who created E by performing a group operation.
$E.\mathsf{packet}$	If E was created honestly, the packet C creating it; else, a special value *'inj'*.
$E.\mathsf{act}$	The group modification performed when E was created: either *'up'* for update, or *'add'*-id_t for adding id_t, or *'rem'*-id_t for removing id_t.
$E.\mathsf{mem}$	The set of group members.
$E.\mathsf{key}$	The shared group key.
$E.\mathsf{chall}$	A flag indicating if a random group key has been outputted.

Pebbling history graph nodes. Besides maintaining the history graph, the $\mathcal{F}_{\text{FR-CGKA}}$ functionality also pebbles and unpebbles the history graph nodes depending on the input operation. In particular, $\mathcal{F}_{\text{FR-CGKA}}$ stores sets Move, Key and $\mathsf{Visited}$. We say that id has a move (resp., key or visited) pebble on U if $(\mathsf{id}, U) \in \mathsf{Move}$ (resp., $(\mathsf{id}, U) \in \mathsf{Key}$ or $(\mathsf{id}, U) \in \mathsf{Visited}$).

Whenever a party id gets corrupted, the history graph is pebbled with *adversarial pebbles* that capture the knowledge gained by the adversary. In particular, all group keys that can be computed by id can also be computed by the adversary. $\mathcal{F}_{\text{FR-CGKA}}$ marks this by putting an adversarial key pebble on each U on which id has a key pebble. Formally, for all $(\mathsf{id}, U) \in \mathsf{Key}$, it adds U to a set $\mathsf{Key}^{\mathcal{A}}$. Second, for each epoch U on which id has a move pebble, the adversary gains the ability to transition to those children V of U for which id does not have a visited pebble on the edge from U to V. This means that the adversary can create children of U on behalf of id and transition to some (honestly created) children of U. $\mathcal{F}_{\text{FR-CGKA}}$ marks this by putting an adversarial move pebble on U and adversarial visited pebbles on some of its children. Formally, for each $(\mathsf{id}, U) \in \mathsf{Move}$, $\mathcal{F}_{\text{FR-CGKA}}$ adds (id, U) to a set $\mathsf{Move}^{\mathcal{A}}$ and for each child V of U such that $(\mathsf{id}, V) \in \mathsf{Visited}$, it adds (id, V) to a set $\mathsf{Visited}^{\mathcal{A}}$.

Inputs from parties. **Initialization** is executed on the first input to $\mathcal{F}_{\text{FR-CGKA}}$. The functionality initializes an empty history graph ($\mathsf{Eps}[*] \leftarrow \perp$) and creates the first epoch using the `*new-ep` helper (see below), with epoch id U_0, chosen by the simulator. It also initializes the sets that store pebbles, described above.

On input `Send`, followed by an epoch id U and action act, $\mathcal{F}_{\text{FR-CGKA}}$ first sends all input values to the simulator and awaits for the acknowledgement flag *ack* (which indicates whether the send operation succeeds or fails with output $\perp$), message C and new epoch id V. In case of success the history graph is updated and a new epoch is created using the helper `*new-ep`. Furthermore, (id, V) is added to Move, Key, $\mathsf{Visited}$, since immediately after sending the message from epoch U, id transitions to V, and is *allowed to move, compute the group key, and has already visited*, V. Finally, the helper `*assert-agree-auth-preserved` is called to enforce agreement of the history graph and authenticity (see below).

On input (`Receive`, c), from id, $\mathcal{F}_{\text{FR-CGKA}}$ first forwards the input values to the simulator and via the first "if" statement checks if the message c corresponds to an existing target epoch id V. In the first case it enforces correctness of the transition to V (from the parent epoch $\mathsf{Eps}[V].\mathsf{par}$) via the helper `*step-correct` (see below), while in the latter, the target epoch V is provided by the simulator. Subsequently, if V already exists, the sender, action and parent epoch are recovered, otherwise, a new epoch is created via `*new-ep`, with inputs provided by the simulator. If act is not for removal, the receiver transitions to the epoch V and Move, Key, $\mathsf{Visited}$, are updated as in `Receive`. Finally, agreement and authenticity are enforced via `*assert-agree-auth-preserved`. On input (`Join`, c), $\mathcal{F}_{\text{FR-CGKA}}$ acts similarly.

On input $(\mathtt{GetKey}, U)$ from id, $\mathcal{F}_{\text{FR-CGKA}}$ outputs the group key of the party with id id. The key is set to a uniformly random value if confidentiality for U holds (this is checked **conf**), otherwise is set to an arbitrary value chosen by the simulator. The operation requires (id, U) to be in the set of key pebbles, Key, and is removed afterwards, which enforces forward secrecy of CGKA protocols.

On input $(\mathtt{DeleteMovePebble}, U)$ from id, (id, U) is removed from Move, indicating that id can no longer move from U to any other epoch.

When id gets corrupted, all pebbles related to id are added to the adversarial sets of pebbles. In particular, for all key pebbles $(\mathsf{id}, U) \in \mathsf{Key}$, U is added to $\mathsf{Key}^{\mathcal{A}}$; for all move pebbles (id, U) not in $\mathsf{Move}^{\mathcal{A}}$, (id, U) is added to $\mathsf{Move}^{\mathcal{A}}$ and similar for $\mathsf{Visited}^{\mathcal{A}}$. The above are only allowed if the corruption is not violating confidentiality of a challenged epoch, while agreement and authenticity are enforced as in previous operations.

Helpers. The `*new-ep` creates a new epoch with creator sndr, parent epoch par, action act, group members mem, and packet packet. `*mem` receives epoch U and action act and updates the set of group members based on the type of action, while `*assert-agree-auth-preserved` enforces agreement and authenticity via the following assertions: (1) all ids visited an epoch U they should belong to the set of group members for that epoch, (2) the history graph has no cycles, (3) `*mem` should update group membership consistently w.r.t. parent epoch and action, and (4) there is no epoch that satisfies the authenticity predicate predicate **auth**, that has been created via an injected packet. Finally, `*step-correct` receives $(\mathsf{Move}, \mathsf{Visited}, \mathsf{id}, V)$ and checks validity of id moving to V, as follows: there should be a move pebble for id w.r.t. the parent of V and id should have not visited V.

Security properties. Intuitively, $\mathcal{F}_{\text{FR-CGKA}}$ captures the security properties of Sec. 3, as follows. Regarding *Agreement*, observe that for each epoch $\mathcal{F}_{\text{FR-CGKA}}$ stores and returns to the caller the same group key, parent epoch, last group modification (action), and group members (set at the time the epoch is created), and if this doesn't hold for the protocol, real and ideal would be easily distinguishable. Furthermore, the 2nd and 3rd assertions inside the `*assert-agree-auth-preserved` enforce consistency in the way that the set of group members is updated w.r.t. the actions issued by the callers, enforcing the same behaviour for the real world protocol. *Confidentiality* is captured by the fact that the adversary is allowed to make calls to the `GetKey` operation, that, for epochs for which confidentiality has not been violated via corruptions (this is checked by the predicate **conf**), the operation returns a uniformly random key (instead of the actual protocol key). This implies that for confidential epochs the actual group key is indistinguishable from a uniformly random value. *Authenticity* is enforced via the last assertion in `*assert-agree-auth-preserved`, which requires that there should be no injected epoch that satisfies the authenticity predicate **auth**. If the assertion

Generic security predicates

$\mathbf{conf}_P(U, \mathsf{Eps}, \mathsf{Move}_0^{\mathcal{A}}, \mathsf{Key}_0^{\mathcal{A}}, \mathsf{Visited}_0^{\mathcal{A}})$ is true if U is a descendant of the epoch created on `Initialization` and there is no sequence of P-valid pebbling steps from $(\mathsf{Move}_0^{\mathcal{A}}, \mathsf{Key}_0^{\mathcal{A}}, \mathsf{Visited}_0^{\mathcal{A}})$ to $(\mathsf{Move}^{\mathcal{A}}, \mathsf{Key}^{\mathcal{A}}, \mathsf{Visited}^{\mathcal{A}})$ such that $U \in \mathsf{Key}^{\mathcal{A}}$.

$\mathbf{auth}_P(V, \mathsf{Eps}, \mathsf{Move}_0^{\mathcal{A}}, \mathsf{Key}_0^{\mathcal{A}}, \mathsf{Visited}_0^{\mathcal{A}})$ is true iff V is a descendant of the epoch created on `Initialization` and there is no sequence of P-valid pebbling steps from $(\mathsf{Move}_0^{\mathcal{A}}, \mathsf{Key}_0^{\mathcal{A}}, \mathsf{Visited}_0^{\mathcal{A}})$ to $(\mathsf{Move}^{\mathcal{A}}, \mathsf{Key}^{\mathcal{A}}, \mathsf{Visited}^{\mathcal{A}})$ s.t. there is a move pebble for the creator $\mathsf{id} = \mathsf{Eps}[V].\mathsf{sndr}$ of V on its parent $U = \mathsf{Eps}[V].\mathsf{par}$.

P-valid pebbling step

Let P be a predicate on a history graph Eps, a party id, an epoch V and a pebbling configuration $(\mathsf{Move}^{\mathcal{A}}, \mathsf{Key}^{\mathcal{A}}, \mathsf{Visited}^{\mathcal{A}})$. A pair of adversarial pebbling configurations $(\mathsf{Move}_0^{\mathcal{A}}, \mathsf{Key}_0^{\mathcal{A}}, \mathsf{Visited}_0^{\mathcal{A}})$ and $(\mathsf{Move}_1^{\mathcal{A}}, \mathsf{Key}_1^{\mathcal{A}}, \mathsf{Visited}_1^{\mathcal{A}})$ on a history graph Eps is P-*valid* if all of the following conditions hold.

1. In addition, move pebbles for id cannot be put on V if id is the sender or is not a member:
$$(\mathsf{id}, V) \in (\mathsf{Move}_1^{\mathcal{A}} \setminus \mathsf{Move}_0^{\mathcal{A}}) \land U \neq \bot \rightarrow \mathsf{id} \in \mathsf{Eps}[V].\mathsf{mem} \land \mathsf{id} \neq \mathsf{Eps}[V].\mathsf{sndr}$$
2. Move pebbles are only put a node V if A) the parent $U = \mathsf{Eps}[V].\mathsf{par}$ already has a move pebble for id and there is no visited pebble on V, or B) $\mathsf{subopt} = P(\mathsf{Eps}, \mathsf{id}, V, \mathsf{Move}_0^{\mathcal{A}}, \mathsf{Key}_0^{\mathcal{A}}, \mathsf{Visited}_0^{\mathcal{A}})$ is true.
$$(\mathsf{id}, V) \in (\mathsf{Move}_1^{\mathcal{A}} \setminus \mathsf{Move}_0^{\mathcal{A}}) \land U \neq \bot \rightarrow \big((\mathsf{id}, U) \in \mathsf{Move}_0^{\mathcal{A}} \land (\mathsf{id}, V) \notin \mathsf{Visited}_0^{\mathcal{A}}\big) \lor \mathsf{subopt}$$
3. Key pebbles are put on nodes V only if a move pebble for some id is put on it:
$$U \in (\mathsf{Key}_1^{\mathcal{A}} \setminus \mathsf{Key}_0^{\mathcal{A}}) \land U \neq \bot \rightarrow \exists \mathsf{id} (\mathsf{id}, U) \in \mathsf{Move}_1^{\mathcal{A}} \setminus \mathsf{Move}_0^{\mathcal{A}}$$
4. Visited pebbles for an id are removed from an edge (U, V) only if a move pebble is put on its parent U:
$$(\mathsf{id}, V) \in (\mathsf{Visited}_0^{\mathcal{A}} \setminus \mathsf{Visited}_1^{\mathcal{A}}) \land U \neq \bot \rightarrow (\mathsf{id}, U) \in \mathsf{Move}_1^{\mathcal{A}} \setminus \mathsf{Move}_0^{\mathcal{A}}$$

Figure 9: Generic security predicates for both our protocols.

fails, the execution halts, making the real and ideal worlds easily distinguishable. *Strong correctness*, is enforced by the fact that for an honestly generated epoch V, $\mathcal{F}_{\text{FR-CGKA}}$ requires (via `*step-correct`) that the receiver has a move pebble on the parent of, and hasn't already visited, V, in which case it always returns the corresponding outputs. This means that for $\mathcal{F}_{\text{FR-CGKA}}$ the receiver should be able to process the incoming message, and if this is not the case for the real world protocol, then real and ideal are easily distinguishable.

3.4 (Sub-)Optimal Security Predicates

We now introduce generic security predicates for confidentiality and authenticity, depicted in Fig. 9, which are used by both our protocols (and potentially other FR-CGKAs). The predicates are parameterized by a protocol-specific predicate P. Intuitively, P identifies epochs which are insecure only due to the protocol's sub-optimal security. That is, P of an optimal FR-CGKA is always false.

First of all, both predicates are false for epochs in detached trees, i.e., not descendants of the root epoch created on `Initialization`. The reason is that, as in [AHKM22a], we do not consider security in detached trees for simplicity. Note that detached epochs may become secure once attached. In the remainder of this section, we do not consider detached epochs.

In general, both predicates are defined as follows. We start with an initial configuration of adversary's pebbles, as recorded by $\mathcal{F}_{\text{FR-CGKA}}$ on corruptions. Then we add more adversary's pebbles, in any way that conforms with pebbling-step P-validity, defined in the bottom part of the figure. An epoch U is confidential, i.e., $\mathbf{conf}(U, \mathsf{Eps}, \mathsf{Move}_0^{\mathcal{A}}, \mathsf{Key}_0^{\mathcal{A}}, \mathsf{Visited}_0^{\mathcal{A}})$ is true, if there is no way we can get to a configuration where $U \in \mathsf{Key}^{\mathcal{A}}$, that is, if the adversary cannot *deduce* a configuration where it knows the key in U. Further, authenticity for V is guaranteed, i.e., $\mathbf{auth}(V,\mathsf{Eps},\mathsf{Move}_0^{\mathcal{A}},\ \mathsf{Key}_0^{\mathcal{A}},\mathsf{Visited}_0^{\mathcal{A}})$ is true if the adversary cannot deduce a configuration with a move pebble for the creator of V on its parent.

Pebbling-step validity. A pebbling step from a configuration $(\mathsf{Eps},\mathsf{Move}_0^{\mathcal{A}},\mathsf{Key}_0^{\mathcal{A}},\mathsf{Visited}_0^{\mathcal{A}})$ to $(\mathsf{Eps},\mathsf{Move}_1^{\mathcal{A}},\ \mathsf{Key}_1^{\mathcal{A}},\mathsf{Visited}_1^{\mathcal{A}})$ is valid w.r.t. the predicate P (or P-*valid*), if rules 1), 2), 3), 4) on Fig. 9 hold. Rules 1), 2) define when the adversary can add a move pebble for id on V, i.e., transition to a configuration such that $(\mathsf{id}, V) \in \mathsf{Move}_1^{\mathcal{A}}$. Intuitively, being able to add a move pebble means that the adversary can deduce secrets known to id in V from the secrets it has given in the first configuration 0. For an optimal protocol, i.e., with $P(\cdot) = \texttt{false}$, this is only possible if the adversary can transition to V by executing id's protocol. This is only possible if 1) id didn't create, and is a member of, V, 2) there is a move pebble on the parent of V for id and id doesn't have a visited pebble on V. Observe that this is the same as the `*step-correct` predicate in $\mathcal{F}_{\text{FR-CGKA}}$. Intuitively, 1) captures PCS — the group heals from id's compromise when id sends a message or is no longer a group member, and 2) captures FS — corrupting id after it transitioned to V does not give the adversary the ability to transition.

The above cases cover what we consider as *optimal* security. Additional, *protocol specific*, weaknesses that enable the adversary to deduce *more* information about the parties' protocol states are captured by the "$\vee\ P(\cdot)$" part of the generic security predicate, which enables the adversary to deduce "more pebbles", by mounting protocol specific attacks that set $P(\cdot)$ to `true`.

4 The FREEK Protocol

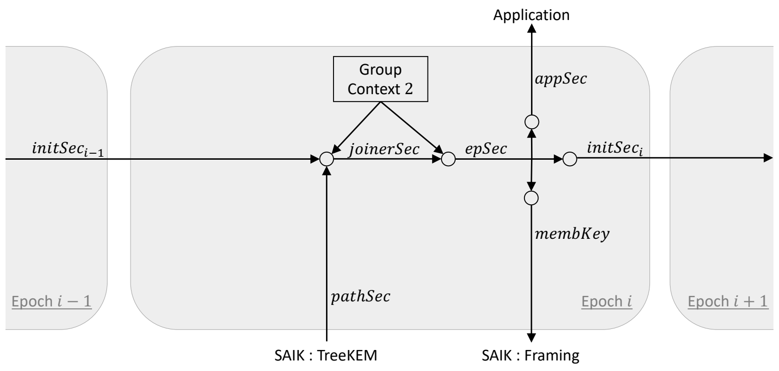
In this section we present the practical FR-CGKA called FREEK. It builds on the SAIK protocol of [AHKM22a], which in turn builds on the MLS protocol of [BBR+22]. We note that all our techniques can be easily applied to MLS too.

Overview of SAIK. For this overview, we distinguish three components of SAIK: *key schedule*, the *TreeKEM protocol* and *message framing*. The key schedule generates for each epoch U a bunch of secrets: the *application*, *membership*, *init* and *epoch*, secrets (and a *joiner*, which is only relevant for adding new members, so we ignore it for now). A secret of U is a κ-bit value known only to the members in U. The *application secret* is the CGKA *group key*. The rest we explain soon.

When a party A creates an epoch V as a child of U, the key schedule generates secrets for V as follows. First, the *init secret* of U is hashed with the *commit secret* of V and the *group context* of V to compute the *epoch secret* of V. The commit secret is a fresh random value which A generates and communicates to all other group members using TreeKEM (TreeKEM guarantees PCS).[6] The context is a cryptographic commitment to all relevant information about V, e.g. the member set and the parent epoch. Finally, all other secrets of V are derived by hashing its epoch secret with different labels.

SAIK's security. The main idea behind the FS and PCS security of SAIK is as follows. Regarding FS, observe that the secrets in V look random (in the RO model) if the adversary does not know the secrets of U, while for PCS, the secrets of V also look random even if the adversary knows the secrets of U, assuming that A generated the commit secret for V, honestly. SAIK also achieves agreement because agreement on the secrets implies agreement on the group context which commits

[6] For the purpose of this overview, one can think of a simplified TreeKEM where each party has a PKE key pair. The sender encrypts a random commit secret to each party and generates a new PKE key pair for themselves.



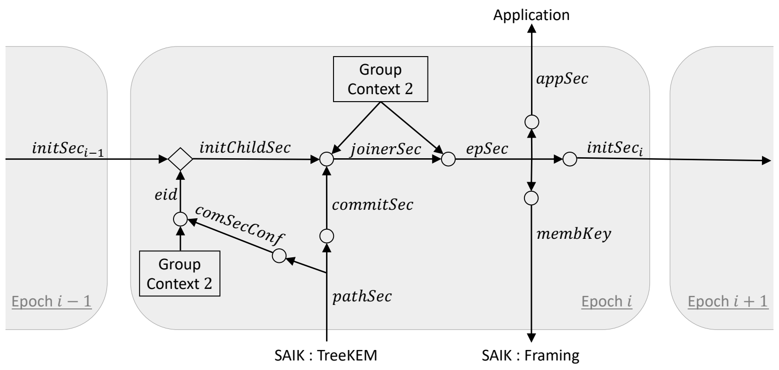


Figure 10: Illustration of the key schedule of SAIK and MLS (top) and of FREEK (bottom). Secrets are represented as arrows and functions are represented as shapes: circles ○ represent KDF calls and diamonds ◇ represent PPRF calls with key coming from the left and input coming from the bottom.

to all information we want the group members to agree on. Finally, SAIK achieves authenticity using the message-framing component. This component equips each member with a signature key pair. When A creates V, her message is framed by signing it with her secret key and MACing with the *membership secret* in U. A's message also includes a new key pair for herself, which provides PCS.

Problems with forks. In SAIK, when a party B transitions from epoch U to V, he immediately deletes the init secret of U. Indeed, this is essential for FS as in this way, if B gets corrupted, say, five epochs later, the secrets of V remain secure, since the adversary does not know the init secret of U. Note that, TreeKEM does not guarantee security of the commit secret of V after the corruption, as the adversary could learn the secret key that decrypts the ciphertext that carries it (this happens if B does not issue an update after epoch V and before getting corrupted). The above creates issues in the presence of forks. For concreteness, say there is a network partition and some party, A', not knowing about A creating V, creates another child of U, say V'. When the connectivity is restored and B learns about V', he would like to derive its secrets (this way he can e.g. read messages encrypted using the CGKA group key in the other partition). However, this is not possible since the same init secret in U protects the group keys in both V and V'. Therefore by deleting it in order to achieve FS of V, B lost the ability to derive the key in V'. A trivial solution to solve this issue would be to *store the old init secrets*, but clearly this completely destroys FS.

Our FR-CGKA protocol. In a high level, our idea to solve the above problem is to modify the key schedule so that it can derive *many init secrets* of an epoch U, one for each child of U. We refer to those secrets as *child init secrets*. That is, the secrets of a child V of U are generated by hashing the *child init secret* (instead of the *init secret* hashed in the old key schedule) of V with its commit secret. After deriving the child init for V, the key schedule deletes all information about it.

The above key schedule can be easily constructed using puncturable PRF, PPRF (we recall the primitive in Sec. 2). In particular, each epoch U has a *parent init secret* which is a PPRF key. To derive the child init for V, the key schedule evaluates the PPRF on input a *challenge* for V (we discuss choosing the challenge soon) and punctures the parent init secret on the challenge.

To summarize, our FR-CGKA protocol works as follows. When B transitions to a child V of U, he generates the secrets of V and stores the entire SAIK state in U, except with the parent init secret punctured on the challenge for V. This means that B keeps one SAIK state for each epoch on which he has a move pebble. To delete a pebble, B simply deletes the state. In this way we get FS: corrupting B after he transitions to a child V does not allow the adversary to re-compute the secrets of V because the parent init of U has been punctured. Moreover, we get fork resilience – if some party A' creates another child V' of U, B can generate its secrets, because the parent init of U has not been punctured on the challenge for V' (and he kept the TreeKEM state).

Choosing the challenge for V. The first idea may be to have the party creating V pick a random challenge and attach it to the packet. However, this increases bandwidth cost and opens the door to denial-of-service (DoS) attacks. For example, say A creates an epoch V as a child of U and picks a random challenge r. Before B sees her packet, a corrupted C creates a child V' of U and sets the challenge for V' to the same r. If B transitions to V' first, he cannot process the honest packet from A because he already punctured the parent init of U on r.

The second idea is to make the challenge be the context of V (recall, the context is a commitment to information about V produced by SAIK). Unfortunately, the context does not bind the commit secret of V which brings back the DoS attack. In particular, the corrupted C can still create V' with the exact same context as V (same group modification etc.) but with a different commit secret. Again, if B transitions to V', he can no longer transition to V.

The third idea is to include both the commit secret and the context in the challenge. However, this is insecure, because a punctured PPRF key generally may reveal the point it was punctured on. Therefore, a corruption of B would reveal old commit secrets via the punctured parent init secrets, destroying FS.

One may try to fix this by setting the challenge to a hash of the commit secret and the context (which is fine if the hash is modeled as an RO). However, this still allows some DoS attacks. At a high level, the problem is that a *removed* member does not know the commit secret of the new epoch (clearly, since they are no longer in the group) and therefore cannot validate the challenge. Concretely, in our running example, say the corrupted C creates V' *by removing B*. After receiving the packet, B punctures his parent init secret on the identifier of V'.[7] B has no way of computing or verifying the identifier, so he must trust C that it is correct. Therefore, C can again claim that the identifier is the same as that of the honest epoch V. After puncturing, B can no longer transition to V.

The final solution is to include in the challenge half of the commit secret and use the other half to generate the secrets of V. That is, let us call the commit secret produced by TreeKEM the path secret (commit is one of path secrets TreeKEM internally generates). The challenge is set to (a hash of) the context and the hash of the path secret with label *'conf'*, called *commit confirmation.* The secrets of V are generated using the new commit secret which is the hash of the path secret with label *'comm'*. Removed members receive the commit confirmation from their removers. Moreover, we choose *epoch*

[7]This is better for FS, because not puncturing on such epochs enables the following attack. A creates epoch 2 as a child of 1 by removing B. B receives her message and later is corrupted. Another member D transitions to a child 3 of 2 and also is corrupted. Now the adversary can combine his secrets with the unpunctured key from B and compromise epoch 2 which would otherwise be secure.

identifiers equal to the challenges. This means that, intuitively, there is a 1-1 correspondence between identifiers and challenges.

This prevents the above attack because B can now compute the epoch identifier the same way as other members. This means that injecting a message creating V' in a way that makes B puncture on an honest identifier V is equivalent to delivering the honest message creating V. The reason is that, as we identify epochs by challenges, the above implies $V = V'$. Further, B can check that the epoch removes him (using the context), just like any other member would.

Note on correctness. Say a party A creates an epoch by sending C. In SAIK the adversary can, without corruptions, successfully deliver different packets c' not outputted by `Ext`, as long as they transition to the honest epoch created by the sender of C. (Trying to make c unique is too stringent, requiring inefficient solutions.) Therefore, it can happen that a party B receives c' first, punctures the parent init and then receives c outputted by `Ext`. Recall that correctness requires that B accepts c. Thus, when receiving a packet claiming to transition to a V (the identifier is included in the packet), B checks if V is already in his state, and returns its stored semantics (while throwing error would be more natural).

Authentication. We use a simplified version of SAIK's framing component. Roughly, each member A generates a signature key pair and updates it each time she sends a message. Further, a secret called the *membership key*, used as a MAC key, is generated by the key schedule for each epoch U. When A creates an epoch V as a child of U, she signs and MACs the identifier V.

We remark that one may ask if puncturing membership keys achieves additional authenticity. Unfortunately, this is not the case. The reason is that this would only prevent forging MAC tags on epoch identifiers of already created epochs. Such forgeries are not useful, however — only forgeries that create new "injected" epochs are an attack on authenticity.

4.1 Definition of FREEK

In the current section we formally define our FR-CGKA protocol, FREEK, based on SAIK [AHKM22b], which, as most CGKA protocols, relies on a PKI infrastructure, called Authenticated Key Service (AKS) in [AHKM22b]. In our protocol description, calls to AKS are made implicitly via calls to SAIK's operations. For completeness we recall the AKS functionality in App. B.

Our protocol relies on the following primitives: (1) a multi-recipient multi-message PKE, mmPKE, (2) a puncturable PRF, PPRF, (3) a signature scheme, Sig, (4) a message authentication code, MAC, and (4) HKDF. Our protocol is depicted in Fig. 11, and includes FREEK's main algorithms (that make calls to SAIK's ones), the key schedule, and the message framing. Due to space limitations, SAIK's unmodified algorithms are deferred to App. B.

The FREEK protocol executed by a party id keeps track of epochs using an array St. For each epoch with identifier U on which id has a move pebble, $\mathsf{St}[U]$ stores a modified SAIK state in that epoch. Further, the protocol keeps track of all outputs of `Join` and `Receive` operations in an array Semantics (cf. the "Note on correctness" paragraph in Sec. 4). Both arrays are initialized when id creates the group using Initialization or joins for the first time. Note that id may join multiple times to different epochs (which is only possible in *FR*-CGKA).

Send. The `Send` operation receives U and act, recovers SAIK's state for epoch U ($\gamma \leftarrow \mathsf{St}[U]$) and uses it to generate a message C and a new state γ' via $\mathsf{SAIK.Send}(\gamma, \mathsf{act})$. Internally (not in Fig. 11) the latter algorithm runs TreeKEM to create and encrypt a value pathSec, which it then inputs to the `*derive-keys` function of the key schedule to compute the secrets of γ'. To enable joining,

FREEK: **Algorithms**

```
Initialization
  $\gamma \leftarrow$ SAIK.Init
  $\mathsf{St}[*] \leftarrow \epsilon$; $\gamma$.eid $\leftarrow 0$; $\mathsf{St}[0] \leftarrow \gamma$
  $\mathsf{Semantics}[*] \leftarrow \bot$
  return $\gamma$.eid

Input (Send, $U$, act $\in$ {'up', 'add'-$\mathsf{id}_t$, 'rem'-$\mathsf{id}_t$}) from id
  $\gamma \leftarrow \mathsf{St}[U]$
  // Run SAIK to get new epoch's state and a packet
  $(\gamma', C) \leftarrow$ SAIK.Send($\gamma$, act)
  $U' \leftarrow \gamma'$.eid; $\mathsf{St}[U'] \leftarrow \gamma$
  // We remove SAIK's signatures and MACs; see App. B
  $\sigma \leftarrow$ *authenticate($\gamma, U'$)
  return $(U', (U, U', C, \sigma, \mathsf{id}, \gamma'.\mathsf{comSecConf}))$

Input (DeleteMovePebble, $U$) from id
  $\mathsf{St}[U] \leftarrow \bot$

Input (Receive, $(U, U', c, \sigma, \mathsf{id}_s, \mathsf{comSecConf})$ from id
  $\gamma \leftarrow \mathsf{St}[U]$
  if $\mathsf{St}[U'] \neq \bot$ then return $\mathsf{Semantics}[U']$
  $(\mathsf{St}[U'], x) \leftarrow$ *process($\gamma, U, U', c, \sigma, \mathsf{id}_s, \mathsf{comSecConf}$)
  $\mathsf{Semantics}[U'] \leftarrow (U, U', x)$
  return $(U, U', x)$

Input (Join, $(U, U', c, \sigma, \mathsf{id}_s, \mathsf{comSecConf})$ from id
  if $\mathsf{St} = \bot$ then $\mathsf{St}[*], \mathsf{Semantics}[*] \leftarrow \bot$
  if $\mathsf{St}[U'] \neq \bot$ then return $\mathsf{Semantics}[U']$
  $(\gamma', x) \leftarrow$ *process($\bot, U, U', c, \sigma, \mathsf{id}_s, \mathsf{comSecConf}$)
  $\mathsf{St}[U'] \leftarrow \gamma'$; $\mathsf{Semantics}[U'] \mathrel{+}\!\leftarrow (U', x)$
  return $(U', x)$

Input (GetKey, $U$) from id
  return SAIK.GetKey($\mathsf{St}[U]$)

helper *process($\gamma, U, U', c, \sigma, \mathsf{id}_s, \mathsf{comSecConf}$)
  if $\gamma \neq \bot$ then try $(\gamma', x) \leftarrow$ SAIK.Receive($\gamma, c$)
  else // SAIK only uses $c$; $U$ is passed to *derive-epoch-keys
      try $(\gamma', x) \leftarrow$ SAIK.Join($c, U$)
  // SAIK didn't return $(\gamma', x) = \bot$ so it succeeded
  req *verify($\gamma, \mathsf{id}_s, U', \sigma$)
  if $\gamma' \neq \bot$ then
      req $U' = \gamma'.\mathsf{eid} \wedge \mathsf{comSecConf} = \gamma'.\mathsf{comSecConf}$
  else // id is removed
      // treeHash$'$ is derived using TreeKEM, $\gamma$ and $c$
      ctx $\leftarrow$ (treeHash$'$, $\mathsf{id}_s$-'rm'-id, $U$)
      req $U' = \mathsf{HKDF.Exp}(\mathsf{ctx}, \mathsf{comSecConf})$
      $\gamma$.initSec $\leftarrow$ PPRF.puncture($\gamma$.initSec, $U'$)
  return $(\gamma', x)$
```

FREEK: **Key schedule**

```
helper *derive-keys($\gamma, \gamma'$, pathSec) // SAIK calls this during Join
and Receive. The input is the old state $\gamma$, the partially initialized
new state $\gamma'$ and pathSec generated by TreeKEM
  $\gamma'$.comSecConf $\leftarrow$ HKDF.Exp(pathSec, 'conf')
  $\gamma'$.eid $\leftarrow$ HKDF.Exp(grpCtxt($\gamma'$), $\gamma'$.comSecConf)
  initChildSec $\leftarrow$ PPRF.eval($\gamma$.initSec, $\gamma'$.eid)
  $\gamma'$.parEid $\leftarrow \gamma$.eid
  joinerSec $\leftarrow$ *derive-joiner($\gamma, \gamma'$)
  $\gamma' \leftarrow$ *derive-epoch-keys($\gamma'$, joinerSec, $\gamma$.parEid)
  return ($\gamma'$, joinerSec)

helper grpCtxt($\gamma$)
  // treeHash (generated by TreeKEM) binds member set
  return ($\gamma$.treeHash, $\gamma$.lastAct, $\gamma$.parEid)

helper *derive-epoch-keys($\gamma'$, joinerSec, $U$)// Called by
*derive-keys and by SAIK.Join with joinerSec received
directly from invitor and $U$ passed from FREEK.*process
  $\gamma'$.parEid $\leftarrow U$
  epSec $\leftarrow$ HKDF.Ext(joinerSec, grpCtxt($\gamma'$))
  $\gamma'$.appSec $\leftarrow$ HKDF.Exp(epSec, 'app')
  $\gamma'$.membKey $\leftarrow$ HKDF.Exp(epSec, 'memb')
  $\gamma'$.initSec $\leftarrow$ HKDF.Exp(epSec, 'init')
  return $\gamma'$

helper *derive-joiner($\gamma, \gamma'$)
  $\gamma$.initSec $\leftarrow$ PPRF.puncture($\gamma$.initSec, $\gamma'$.eid)
  commitSec $\leftarrow$ HKDF.Exp(pathSec, 'com')
  return HKDF.Ext($\gamma'$.grpCtxt(), initChildSec, commitSec)
```

FREEK: **Message framing**

```
helper *authenticate($\gamma, x$)
  return (MAC.tag($\gamma$.membKey, $x$), Sig.sign($\gamma$.ssk, $x$))

helper *verify($\gamma, \mathsf{id}_s, x, (\sigma, t)$)
  // In SAIK leafof($\mathsf{id}_s$).spk is $\mathsf{id}_s$'s personal public key
  return $\leftarrow$ MAC.vrf($\gamma$.membKey, $x, t$) $\wedge$
  Sig.vrf($\gamma$.leafof($\mathsf{id}_s$).spk, $x, \sigma$)
```

Figure 11: FREEK algorithms.

SAIK also encrypts the joinerSec computed in `*derive-keys` to all new members (they cannot run `*derive-keys`, since they should not know the init secret — this may allow them to compute group keys for other forks where they are not members).

We modify `*derive-keys` as follows; this is one of the core changes made by FREEK. The function computes a distinct init secret for γ', denoted by initChildSec. This value is the output of PPRF.eval over the PPRF key, initSec, of γ, and the epoch id of γ', where the latter is the output of HKDF.Exp over the group context, $\mathsf{grpCtxt}(\gamma')$, and the confirmation secret, $\mathsf{comSecConf}$. The joiner's secret, $\mathsf{joinerSec}$, is now computed w.r.t. the group context, $\mathsf{grpCtxt}()$, $\mathsf{initChildSec}$ and the commit secret, $\mathsf{commitSec}$, and from $\mathsf{joinerSec}$, all other secretes are derived as in SAIK. Obverse that the PPRF key in γ is punctured over the id of γ', and this enables FS, i.e., if the user's PPRF key is leaked after processing the incoming message creating γ', the adversary can no longer evaluate the PPRF over its identifier and compute the $\mathsf{initChildSec}$.

The rest of the Send operation fetches from γ' the epoch identifier U created by `*derive-keys` and stores the state in St. Finally, it MACs and signs U (SAIK MACs and signs a different value which is not useful for us.)

Receive and join. `Receive`, on input $(U, U', c, \sigma, \mathsf{id}_s, \mathsf{comSecConf})$, first recovers the SAIK's state that will be used to process the incoming message for epoch U ($\gamma \leftarrow \mathsf{St}[U]$) and if id has already processed a message that leads to the target epoch, U', the operation returns the values stored in $\mathsf{Semantics}[U']$ (computed the first time id processed a message that led to U'). If not, it processes the message by executing SAIK's `Receive` or `Join` operation with c. Internally, (not in Fig. 11) SAIK's `Receive` algorithm runs TreeKEM to decrypt $\mathsf{pathSec}$ which it inputs to our modified `*derive-keys` function to get the key schedule for γ'. Similarly, SAIK's `Join` operation decrypts $\mathsf{joinerSec}$ and runs `*derive-epoch-keys`.

Observe that the protocol checks if the sender knows the right epoch id, U', and if the user is removed it directly uses $\mathsf{comSecConf}$ to do that check. Removed users also puncture the init secret on the epoch from which they are removed, U'. This prevents the adversary from injecting a message that removes id from U', so that later id cannot transition to an honestly generated epoch U'.

Security of FREEK is proven in Sec. 7.

5 FR-CGKA with Optimal Security

In this section we give an overview of the protocol $\mathsf{O\text{-}FREEK}$, that achieves *optimal security* predicates, i.e., the predicates from Fig. 9 with P set to false.

Roughly, we start with a simplified version of SAIK called $\mathsf{SAIK\text{-}S}$ which is less efficient but just as secure. We then identify and fix two problems with it: cross-fork and collusion attacks. The first fix uses a known technique from [ACJM20], using HIBE. The second fix requires modifying the HIBE fix to *also* deal with new attacks specific to FR-CGKA.

The $\mathsf{SAIK\text{-}S}$ protocol. In $\mathsf{SAIK\text{-}S}$, each member has a PKE key pair. When an epoch is created, the committer encrypts a random path secret to all members and the new epoch's secrets are derived by mixing it with the (child) init secret. It also generates a new PKE (and signature) key pair for itself.

Weak PCFS and cross-fork attacks. Cross-fork attacks are relevant for CGKA in general (not only FR-CGKA). Roughly, the problem is that parts of the private state of parties in different epochs may be the same. In that case, the same secret may be relevant to two (or more) distinct epochs, and it is impossible to meaningfully delete a move pebble on one epoch but not another.

For concreteness, assume that a protocol execution creates an epoch 1 on which parties A, B and C all have a move pebble. Then a fork is created as follows: A creates (and transitions to) epoch 2 and at the same time B creates epoch 3. Notice that C's PKE key is the same in epochs 1, 2 and 3. Now assume that C transitions to epoch 2, *deletes the move pebble on 1* and then gets corrupted. The adversary can use C's PKE secret key to decrypt the commit secret in epoch 3.

Intuitively, to defend against cross-fork attacks, a protocol needs to ratchet *all key material (of all group members), with each epoch change.* This has to be done in a way such that, secrets in any epoch V do not reveal information about its *ancestors or epochs in other arms of a fork.* Following [ACJM20], we achieve this using HIBE. Roughly, the public key of each party in an epoch U is a pair of a HIBE (master) public key and a vector of HIBE identities. The latter contains the identifiers of all epochs since the key was inserted into the state, ordered from the oldest to U. When a party A creates a child epoch of U, it encrypts the fresh commit secret to the HIBE public key and identity vector of each (non-removed) member B in U. Then A computes the public group state in V as follows: the identifier V is the same as in SAIK-S; the public key of each other group member B is the HIBE public key from U and the identity vector from U with V appended; the public key of A is a fresh HIBE master key she chooses and the empty identity vector. When B transitions to V, it updates the HIBE secret key of U with V and stores the resulting secret key that will be used to process incoming messages for V; it also keeps the HIBE key before the update. When later B removes the move pebble on U it simply erases that key (i.e., the key before the update).

At a high level, the HIBE secret key of an epoch U enables key generation only for the descendants of U. This prevents the attack described above since all secrets in an epoch V are (computationally) unrelated to the secrets in any epochs other than the descendants of V.

Collusion attacks. This type of attacks is specific to the way SAIK-S punctures the init secrets to get forward-secure FR-CGKA. In particular, observe that in SAIK-S, if a party A has a move pebble on an epoch U, then it can decrypt the commit secrets of all children of U, even those on which it has visited pebbles. The only thing that protects the children in case A is corrupted is the fact that A punctured the init secret on their identifiers. However, the unpunctured init can be leaked when other members are corrupted, which enables *collusion attacks.*

For concreteness, assume that in a protocol execution, A, B and C, are in epoch 1. Then, A creates (and transitions to) epoch 2 by removing C; B transitions to 2, creates epoch 3 as its child, and deletes the group key in epoch 2. He now has move pebbles on epochs 1, 2 and 3, and visited pebbles on 2 and 3. C never sees epoch 2. Now assume B and C get corrupted. The expectation is that epoch 2 is secure since C cannot move there because A removed him, and B has a visited pebble on it and deleted the group key. Unexpectedly, epoch 2 is insecure for the following reason: the adversary can compute the group key in 2 by combining the unpunctured init secret from C and the commit secret decrypted using B's secret key from epoch 1.

Intuitively, puncturing the init secret is insufficient, since it is known to other group members, and the adversary can leak the (unpunctured) key via corruptions of removed users, or group members that lie in other epochs. Therefore, in order for security to be preserved in the presence of collusion attacks we require the following: when B puts a visited pebble on V with parent U, he has to puncture *his own* secret key in U, so that it can no longer decrypt the ciphertext that carries the commit secret of V.

To achieve this, we modify the HIBE-version of SAIK-S as follows. The main observation is that HIBE with a binary identity space, called binary-tree encryption, BtPke, (a special case of HIBE [GS02]) can be used to construct puncturable public key encryption [GM15], whose goal is exactly to allow A to puncture her secret key. So, we first switch from HIBE to BtPke. Second, we switch the order of encryption and updating the identity vector: when A creates a new epoch V, then for each other member B, she first appends all bits of V to B's identity vector $\vec{U}$ in U and then encrypts the commit secret to B's public key and the identity $\vec{V} = \vec{U} \parallel V$. So, the commit is encrypted to its unique identity vector $\vec{V}$. When later B puts a visited pebble on V, he punctures his secret key in U on the unique $\vec{V}$, as we describe below.

Puncturing in our setting is essentially a public-key version of puncturing a GGM tree. That is,

B's secret key in U is a binary tree $\mathsf{eskTree}_U$ of height $|V|$, where each node is labeled by a binary identity vector $\vec{A}$. The root's label is $\vec{U}$ and if a node has label $\vec{A}$, then its left and right children have labels $\vec{A} \parallel 0$ and $\vec{A} \parallel 1$, respectively. In addition, some nodes store the HIBE secret keys corresponding to their labels. We denote the HIBE secret key of a node $\vec{U} \parallel \vec{A}$ by $\mathsf{eskTree}_U[\vec{A}]$. Initially, when U is created, only the root stores a key in $\mathsf{eskTree}_U[\epsilon]$. Notice that $\vec{V} = \vec{U} \parallel V$, is a label on a leaf in $\mathsf{eskTree}_U$. To puncture the tree on $\vec{V}$, B derives and stores HIBE secret keys for all nodes on the copath of that leaf to the root and then deletes all nodes in the path to the root.

Fixing the last issues. As described so far, after puncturing $\mathsf{eskTree}_U$, B creates the tree $\mathsf{eskTree}_V$ for the new epoch V with the root set to $\vec{V} = \vec{U} \parallel V$. However, this creates an issue because the HIBE secret key for $\vec{V}$ corresponds to the public key used by A to encrypt the commit secret of V. Thus, storing the secret key breaks FS. To fix this, B stores instead the secret for the left child of $\vec{V}$, i.e. $\vec{V} \parallel 0$. Respectively, A and all other members update the identity vectors with 0 after encrypting. Further, the protocol no longer needs the init secret, since puncturing HIBE keys solves all problems it was meant to solve, namely, getting forward secrecy and being able to put visited pebbles on nodes.

5.1 Authenticity

Cross-fork attacks. Cross-fork attacks apply to both confidentiality and authenticity. For concreteness, recall the protocol execution showcasing cross-fork attacks on confidentiality: A, B and C each have a move pebble on an epoch 1. Then a fork occurs: A creates epoch 2 and at the same time B creates epoch 3. C transitions to 2 but does not know about 3 and deletes the move pebble on 1. Notice that C's signature key is the same in epochs 2 and 3. Therefore, corrupting him now allows the adversary to create an epoch 4 as a child of 3 by injecting a message to B on behalf of C. This is an attack because C does not have a move pebble on 3.

One attempt to solve this would be to MAC packets with some shared group secrets (the way FREEK does). This does not work, because the adversary can obtain MAC keys by corrupting B. Note that creating 4 is still an attack because the adversary does not have a move pebble *for* C on 3.

The authors of [ACJM20] propose to prevent cross-fork attacks for regular CGKA using hierarchical identity-based signatures, HIBS. The HIBS primitive is analogous to HIBE — verification is done with respect to a vector of identities indicating a place in the hierarchy and a secret signing key can be used to derive secret keys below it. Signing messages in the optimally secure CGKA of [ACJM20] also works analogously to encryption. At a high level, recall that each party A in an epoch U has a HIBE master public key and a vector of identities of ancestors of U. To add optimal authenticity, A also has a HIBS master verification key. Messages sent by A are verified with that key and the same identity vector as for HIBE and A generates a new master key for herself every time she sends a message.

Translating this directly to the FR-CGKA setting means that a party has one HIBS secret key for each epoch U on which it has a move pebble. The key for U corresponds to the identifiers of the ancestors of U since it was generated. The cross-fork attack describe above is prevented because corrupting C reveals only the signing key for identity vector $(1, 2)$ which does not allow the adversary to inject messages in 3, since they are verified with the vector $(1, 3)$.

Deleting move pebbles. Unfortunately, the FR-CGKA sketched above is still not optimally secure. At a high level, the problem is that it is not possible to delete a move pebble on a child V of U while keeping the move pebble on U. In particular, deleting the HIBS secret key for V doesn't solve our problem, since it can be derived from the HIBS secret key for U.

O-FREEK: Algorithms

```
Initialization
  if $\mathsf{id} = \mathsf{id}_{\mathsf{creator}}$ then
      $(\mathsf{epk}, \mathsf{esk}) \leftarrow \mathsf{BtPke.gen}()$
      $(\mathsf{spk}, \mathsf{ssk}) \leftarrow \mathsf{BtSig.gen}()$
      $\gamma \leftarrow$ *init-state$(\mathsf{esk}, \mathsf{ssk}, \{(\mathsf{id}, \mathsf{epk}, \mathsf{spk})\})$
      $\gamma.\mathsf{appSec} \xleftarrow{\$} \{0,1\}^\kappa$

Input $(\mathtt{Send}, U, \mathsf{act})$ from id
  req $\mathsf{St}[U] \neq \bot$
  // Initialize the new epoch's state $\gamma$
  try $(\gamma, \mathsf{act}) \leftarrow$ *apply-act$(\gamma[U].\mathsf{clone}(), \mathsf{id}, \mathsf{act})$
  *update-context$(\gamma, U, \mathsf{id}, \mathsf{act})$
  $r \xleftarrow{\$} \{0,1\}^\kappa$
  $\mathsf{comSecConf} \leftarrow \mathsf{CR\text{-}PRF}(r, \text{'}conf\text{'})$
  $\gamma.\mathsf{appSec} \leftarrow \mathsf{CR\text{-}PRF}(r, \text{'}app\text{'})$
  $V \leftarrow \mathsf{Hash}(\gamma.\mathsf{grpCtxt}, \mathsf{comSecConf})$
  // Update "identity chains" in $V$ and encrypt $r$
  for $\mathsf{id}' \in \gamma.\mathsf{BtIds} \setminus \{\mathsf{id}\}$ do
      $\mathsf{btid} \leftarrow \gamma.\mathsf{BtIds}[\mathsf{id}'] \parallel V$
      $C[\mathsf{id}'] \leftarrow \mathsf{BtPke.enc}(\gamma.\mathsf{ePKs}[\mathsf{id}'], \mathsf{btid}, r)$
      $\gamma.\mathsf{BtIds}[\mathsf{id}'] \leftarrow \mathsf{btid} \parallel 0$
  $\mathsf{St}[V] \leftarrow \gamma$
  // Sign $V$ and update the signing key
  $(\sigma, \mathsf{St}[U].\mathsf{sskTree}, \mathsf{ssk}) \leftarrow$ *sign$(\mathsf{St}[U].\mathsf{sskTree}, V, V)$
  return $(V, (C, U, V, \mathsf{id}, \mathsf{act}, \mathsf{comSecConf}, \sigma))$
```

```
Input $(\mathtt{Receive}, (c, U, V, \mathsf{id}_s, \mathsf{act}, \mathsf{comSecConf}, \sigma))$ from id
  try $(\mathsf{St}[V], \mathsf{St}[U].\mathsf{eskTree}, \mathsf{St}[U].\mathsf{sskTree})$
          $\leftarrow$ *process$(\mathsf{St}[U], c, U, V, \mathsf{id}_s, \mathsf{act}, \mathsf{comSecConf}, \sigma)$
  if $\mathsf{act} =$ 'add'-$\mathsf{id}_t$-$(\mathsf{pk}_t, \mathsf{spk}_t)$ then
      return $(\mathsf{id}_s,$ 'add'-$\mathsf{id}_t)$
  else return $(\mathsf{id}_s, \mathsf{act})$

Input $(\mathtt{Join}, (c, U, V, \mathsf{id}_s, G, \sigma))$ from id
  // $G$ is the group roster computed by the DS
  req $(\mathsf{id}, \mathsf{epk}, \mathsf{spk}) \in G$ for some $\mathsf{epk}, \mathsf{spk}$
  $(\mathsf{esk}, \mathsf{ssk}) \leftarrow$ query $(\mathtt{GetSk}, (\mathsf{epk}, \mathsf{spk}))$ to $\mathcal{F}_{\text{KS}}$
  $\gamma \leftarrow$ *init-state$(\mathsf{esk}, \mathsf{ssk}, G)$
  $\mathsf{act} \leftarrow$ 'add'-id-epk-spk
  $(\gamma, \_, \_) \leftarrow$ *process$(\gamma, c, U, V, \mathsf{id}_s, \mathsf{act}, \sigma)$
  req $G$ is the set of keys in $\gamma.\mathsf{ePKs}$ and $\gamma.\mathsf{sPKs}$
  $\mathsf{St}[*] \leftarrow \bot$; $\mathsf{St}[V] \leftarrow \gamma$
  return $(V, \mathsf{id}_s, G)$

Input $(\mathtt{DeleteMovePebble}, U)$ from id
  $\mathsf{St}[U] \leftarrow \bot$

Input $(\mathtt{GetKey}, U)$ from id
  $(\mathsf{appSec}, \mathsf{St}[U].\mathsf{appSec}) \leftarrow (\mathsf{St}[U].\mathsf{appSec}, \bot)$
  return appSec
```

Figure 12: Algorithms of O-FREEK.

Intuitively, to fix this, we have to *puncture* the HIBS secret key in U so that it is no longer possible to derive the HIBS secret for V. To this end, we use a HIBS with a binary identity space, called binary-tree signature, BtSig (we recall the formal definition in Sec. 2). Puncturing works analogous to puncturing BtPke secret keys. That is, the signing key for U is a binary tree sskTree with the same structure as the secret-key tree eskTree for BtPke — nodes have the same labels and whenever a node in eskTree stores a BtPke key for an identity vector $\vec{I}$, the same node in sskTree stores a BtSig key for $\vec{I}$.

5.2 The Final Construction

The protocol is defined Figs. 12 and 13. It relies on the following primitives: (1) a CCA secure binary-tree encryption scheme, BtPke, (2) an EUF-CMA binary-tree signature scheme, BtSig, (3) a collision resistant PRF, CR-PRF, and (4) collision resistant hash functions, Hash. We now briefly explain the protocol operations depicted in the below figures.

`Initialization` generates keys for binary-tree encryption and signatures, initializes the user's state with the generated keys and samples the first group key, appSec. `Send`, on input epoch id U and action act, applies the action to a copy of U's state, updates the group context, and samples uniformly random r from which it generates the confirmation secret, comSecConf and application secret appSec, and the epoch id V for the new epoch (which is the hash of comSecConf). It then updates the identity chains in V and encrypts r to all group members. Finally, it signs the new epoch id, V, updates the signing key by puncturing sskTree, and generates the signing key for V by ratcheting forward ssk. `Receive`, on input $(c, U, V, \mathsf{id}_s, \mathsf{act}, \sigma)$, verifies the signature over V, initializes the state and updates the group context for the new epoch, decrypts c and derives the secret key for V by ratcheting forward esk, computes secrets as in `Send`, verifies V via Hash, and updates the

O-FREEK: Helpers

```
helper *apply-act($\gamma$, $\mathsf{id}_s$, act)
  // If necessary, get keys for the action
  if act = 'up' then // My own update
      (epk, esk) ← BtPke.gen()
      (spk, ssk) ← BtSig.gen()
      $\gamma$.BtIds[id] ← $\epsilon$
      $\gamma$.eskTree ← [*] $\gamma$.eskTree[$\epsilon$] ← esk
      $\gamma$.sskTree ← [*] $\gamma$.sskTree[$\epsilon$] ← ssk
      act ← 'up'-epk-spk
  if act = 'add'-$\mathsf{id}_t$ then
      ($\mathsf{epk}_t$, $\mathsf{spk}_t$)
                ← query (GetPk, $\mathsf{id}_t$) to $\mathcal{F}_{\text{KS}}$
      act ← 'add'-$\mathsf{id}_t$-($\mathsf{epk}_t$, $\mathsf{spk}_t$)
  // Apply the action
  if act = 'up'-epk-spk then
      ($\gamma$.ePKs[$\mathsf{id}_s$], $\gamma$.sPKs[$\mathsf{id}_s$]) ← (epk, spk)
  else if act = 'add'-$\mathsf{id}_t$-epk-spk then
      ($\gamma$.ePKs[$\mathsf{id}_t$], $\gamma$.sPKs[$\mathsf{id}_t$]) ← (epk, spk)
  else if act = 'rem'-$\mathsf{id}_t$ then
      ($\gamma$.ePKs[$\mathsf{id}_t$], $\gamma$.sPKs[$\mathsf{id}_t$]) ← ($\perp$, $\perp$)
  return ($\gamma$, act)

helper *sign(sskTree, $V$, $m$)
  try (sskTree$'$, ssk)
          ← *punct(sskTree, $V$, BtSig.der)
  try $\sigma$ ← BtSig.sig(ssk, $m$)
  // Get new root for $V$ by ratcheting ssk
  (ssk, _) ← BtPke.der(ssk)
  return ($\sigma$, sskTree$'$, ssk)
```

```
helper *process($\gamma$, $c$, $U$, $V$, $\mathsf{id}_s$, act,
                                        comSecConf, $\sigma$)
  (spk, btid) ← ($\gamma$.sPKs[$\mathsf{id}_s$], $\gamma$.BtIds[$\mathsf{id}_s$])
  req BtSig.ver(spk, btid, $V$, $\sigma$)
  // Compute the new epoch's state
  try $\gamma'$ ← *apply-act($\gamma$.clone(), $\mathsf{id}_s$, act)
  *update-context($\gamma'$, $\mathsf{id}_s$, act)
  if act = 'rem'-id then // Leave group
      ($\tau$, _) ← *punct($\gamma$.tree, $V$, BtPke.der)
      // The new state in $U$ is $\perp$
      return ($\perp$, $\tau$, $\gamma$.sskTree)
  // Derive new BtPke secret, decrypt $c$.
  try ($r$, eskTree, esk)
              ← *decrypt($\gamma$.eskTree, $V$, $c$)
  comSecConf ← CR-PRF($r$, 'conf')
  $\gamma'$.appSec ← CR-PRF($r$, 'app')
  // Check that $V$is correct
  req $V$ = Hash($\gamma'$.grpCtxt, comSecConf)
  // Packet accepted.
  $\gamma'$.eskTree ← [*]; $\gamma$.eskTree[$\epsilon$] ← esk
  (_, sskTree, ssk)
          ← *sign(St[$U$].sskTree, $V \parallel 0$, $\epsilon$)
  $\gamma'$.sskTree ← [*]; $\gamma$.sskTree[$\epsilon$] ← ssk
  return ($\gamma'$, eskTree, sskTree)

helper *init-state(esk, ssk, $G$)
  $\gamma$.esk[*] ← $\perp$; $\gamma$.esk$\epsilon$] ← esk
  $\gamma$.ssk[*] ← $\perp$; $\gamma$.ssk$\epsilon$] ← esk
  for (id, epk, spk) ∈ $G$ do
      ($\gamma$.ePKs[id], $\gamma$.sPKs[id], $\gamma$.BtIds)
                                    ← (epk, spk, $\epsilon$)
  return $\gamma$
```

```
helper *punct(tree, $\vec{I}$, *deriver)
  // Derive the secret key tree[$V$] and at the
  same time puncture tree on $V$
  tree ← tree.clone()
  for $i = 0$ to $|\vec{I}|$ do
      $\vec{I_i}$ ← $\vec{I}[0..i-1]$
      if tree[$\vec{I_i}$] $\neq \perp$ then
          (tree[$\vec{I_i} \parallel 0$], tree[$\vec{I_i} \parallel 1$])
                              ← *deriver(tree[$\vec{I_i}$])
          tree[$\vec{I_i}$] ← $\perp$
  // If tree hasn't been punctured on $\vec{I}$ then
  the loop set the key for $\vec{I}$.
  req tree[$\vec{I}$] $\neq \perp$
  (sk, tree[$\vec{I}$]) ← (tree[$\vec{I}$], $\perp$)
  return (tree, sk)

helper *update-context($\gamma$, $U$, $\mathsf{id}_s$, act)
  $\gamma$.grpCtxt ← Hash($\gamma$.ePKs, $\gamma$.sPKs, $U$, $\mathsf{id}_s$, act)

helper *decrypt(eskTree, $V$, $c$)
  try (eskTree$'$, esk)
          ← *punct(eskTree, $V$, BtPke.der)
  try $r$ ← BtPke.dec(esk, $c$)
  // Get new root for $V$ by ratcheting
  forward esk
  (esk, _) ← BtPke.der(esk)
  return ($r$, eskTree$'$, esk)
```

Figure 13: The helpers for O-FREEK.

signing key. `Join` is similar, while `DeleteMovePebble` and `GetKey` are straightforward.

Remark 1. *In general, our protocol requires BtPke with unbounded hierarchy depth and adaptive security, which is quite a strong requirement. Using BtPke with a bounded hierarchy would come at the cost of sacrificing correctness — it would result in an upper bound on the number of epochs that can be created while there is a single party that never sends an update. We note that this may be reasonable in some applications, in which the silent parties can be removed.*

Requiring adaptive security seems necessary to get adaptively-secure FR-CGKA. We note that in the random oracle model there is a trivial black-box construction of adaptively-secure BtPke from statically-secure BtPke — the adaptively-secure construction simply hashes identities via the RO before passing them to the statically-secure one. Therefore, adaptive security is only a limitation if we require security in the standard model.

Security of O-FREEK is proven in Sec. 8.

6 Natural Fork-Resolution Protocols

In this section, we discuss how any FR-CGKA can be used together with *a natural fork-resolution protocol* to get a complete solution to group management. At a high level, the goal of a natural fork-resolution protocol is to allow group members whose views of the group have diverged to reconcile

Protocol FR-Wrap

```
Initialization // Executed on first input.
    $U \leftarrow \mathcal{F}_{\text{FR-CGKA}}.\texttt{Initialize}$
    $G \leftarrow \{U\}$
    $E, \mathsf{Move}, \mathsf{Visited} \leftarrow \varnothing$
    $A[*], M[*] \leftarrow \bot$

Input GetGraph from id
    return $(G, E, A, M, \mathsf{Move}, \mathsf{Visited})$

Input $(\texttt{ExtendGraph}, G', E', A', M', \mathsf{Move}', \mathsf{Visited}')$ from id
    $G \mathrel{+}\leftarrow G'$; $E \mathrel{+}\leftarrow E'$; $A \mathrel{+}\leftarrow A'$
    $\mathsf{Move} \mathrel{+}\leftarrow \mathsf{Move}'$; $\mathsf{Visited} \mathrel{+}\leftarrow \mathsf{Visited}'$
    req $(G, E)$ is a forest

Inputs $(\texttt{CreateNode}, U, \mathsf{act})$, $(\texttt{Send}, U, \mathsf{act})$ from id
    $(V, C) \leftarrow \mathcal{F}_{\text{FR-CGKA}}.\texttt{Send}(U, \mathsf{act})$
    $*\texttt{register}(U, V, \mathsf{id}, \mathsf{act}, C)$
    $\mathsf{Move}, \mathsf{Visited} \mathrel{+}\leftarrow (\mathsf{id}, V)$
    return $(V, C)$

Inputs $(\texttt{CreateEdge}, c)$, $(\texttt{Receive}, c)$ from id
    $(U, V, \mathsf{id}_s, \mathsf{act}) \leftarrow \mathcal{F}_{\text{FR-CGKA}}.\texttt{Receive}(c)$
    $*\texttt{register}(U, V, \mathsf{id}_s, \mathsf{act}, \bot)$
    $\mathsf{Move}, \mathsf{Visited} \mathrel{+}\leftarrow (\mathsf{id}, V)$
    return $(U, V, \mathsf{id}_s, \mathsf{act})$

Input $(\texttt{DeleteMovePebble}, U)$ from id
    return $\mathcal{F}_{\text{FR-CGKA}}.\texttt{DeleteMovePebble}(U)$

Input $(\texttt{Join}, c)$ from id
    $(U, \mathsf{id}_s, \mathsf{mem}) \leftarrow \mathcal{F}_{\text{FR-CGKA}}.\texttt{Join}(c)$
    $G \mathrel{+}\leftarrow U$
    $M[U] \leftarrow \mathsf{mem}$
    $\mathsf{Move}, \mathsf{Visited} \mathrel{+}\leftarrow (\mathsf{id}, U)$
    return $(U, \mathsf{id}_s, \mathsf{mem})$

Input $(\texttt{GetKey}, U)$ from id
    return $\mathcal{F}_{\text{FR-CGKA}}.\texttt{GetKey}(U)$

helper $*\texttt{register}(U, V, \mathsf{id}_s, \mathsf{act}, C)$
    $G \mathrel{+}\leftarrow U, V$; $E \mathrel{+}\leftarrow (U, V)$
    $A[(U, V)] \leftarrow (\mathsf{id}_s, \mathsf{act}, C)$
```

Figure 14: The protocol FR-Wrap supporting fork-resolution protocols.

these views and define *the current* group state. A bit more precisely, a group member can use the fork-resolution protocol to collect information about the group's history graph from other members (assuming point-to-point channels) and identify (or create) an epoch representing the current state consistent with the collected view.

The FR-Wrap protocol. To support fork-resolution protocols, we construct a simple protocol called FR-Wrap, defined in Fig. 14. The protocol works in the $\mathcal{F}_{\text{FR-CGKA}}$-hybrid model, that is, it makes black-box calls to $\mathcal{F}_{\text{FR-CGKA}}$ and can be instantiated with any FR-CGKA protocol that realizes it. FR-Wrap extends the interface of $\mathcal{F}_{\text{FR-CGKA}}$ by methods especially for fork-resolution protocols. That is, in addition to all inputs to $\mathcal{F}_{\text{FR-CGKA}}$ (which are transparently forwarded to the functionality), FR-Wrap supports the following.

First, a fork-resolution protocol can fetch the labeled history graph (input GetGraph), represented by a set G of epoch identifiers and a set E of edges. Each edge is labeled by the action creating it, stored in the array A. Some nodes are labeled by the member sets, stored in the array M. This is important for epochs with unknown parents, such as ones into which the graph's owner joined. Finally, the output contains move and visited pebbles about which the graph's owner knows. In the following, by *graph* we refer to the tuple $(G, E, A, M, \mathsf{Move}, \mathsf{Visited})$.

Second, FR-Wrap supports collecting information about the graph from other members by allowing to extend the graph by another one, obtained from the other member (input ExtendGraph). This method involves no cryptography and does not affect $\mathcal{F}_{\text{FR-CGKA}}$ — a party does not want to download packets and transition to epoch (which is expensive) if this is not useful given the eventual current group state. Finally, FR-Wrap supports creating the current epoch (in case it does not exist) by allowing to *create* nodes and edges (inputs CreateNode, CreateEdge). Creating a node has the effect of creating an epoch in $\mathcal{F}_{\text{FR-CGKA}}$, and creating an edge has the effect of transitioning to an epoch in $\mathcal{F}_{\text{FR-CGKA}}$. The latter requires obtaining a packet c (either from some server, or some other party).

Typical fork-resolution protocols. A typical fork-resolution protocol uses FR-Wrap as follows. The protocol is invoked whenever an id sends a message to id′ and they discover that the current epoch is different according to their views. That is, id sends from a different epoch. (We assume here that parties communicate via a mesh network, so they are both online at this moment; we discuss other cases later.) Recall that the goal is for id and id′ to transition to a new current epoch, consistent with the union of their views. The protocol proceeds as follows:

1. Each id and id′ fetches its graph (GetGraph) and sends it to the other party, who adds it to its own graph via ExtendGraph.
2. id (resp., id′) inputs its current graph to a local *state-resolution algorithm* (see below) which outputs the current set of members M consistent with it.
3. id creates a new epoch with member set M. To this end, id picks some node U (see below) with member set M' and creates a chain starting at U that "corrects" M'. That is, for party in $M' \setminus M$, id *creates* a node removing them (CreateNode). Similarly, for each party in in $M \setminus M'$, id creates a node adding them. Finally, id identifies all members who are in U but cannot transition there based on Move and Visited. It removes and re-adds each such party.[8]
4. id sends to id′ the FR-CGKA messages generated by edge creation. id′ transitions to the new current epoch via CreateEdge.

State-resolution algorithms. The high-level goal of a state-resolution algorithm is to "make sense" of a number of concurrent group changes and output "the right" group state. One example is the State Resolution v2 algorithm of [Fou23a].

At a very high level, [Fou23a] represents the history of a group as a collection of events, which are roughly the same thing as our group operations. In fact, [Fou23a] defines much more event types (including state events that do not influence members' permissions) but here we focus on those we have in common (it is not hard to extend our FR-CGKA by other types): adds, removes, joins and group creation ([Fou23a] does not have special update events). Relations between control events are represented as a directed acyclic graph (DAG). An edge from event E to E' indicates that a party generating E' knew about E at the time (in the protocol of [Fou23a], this relation is made explicit in the packet).[9]

State Resolution v2 takes as input an event DAG and resolves the "current" room state. It achieves this by first topologically sorting the DAG, which gives a linear sequence of events. Then, the algorithm initializes a group state according to the first event (joining or group creation) and traverses the control-event sequence one by one. For each event, if it is *not* authorized according to the current state, it removes it from the sequence. Else, it updates the current state according to it. After the last event, we get the current state.

We can use an analogous algorithm to implement state resolution in our sense. That is, the history graph from FR-Wrap is first topologically sorted, which gives a sequence of group operations. Then, a membership set is initialized according to the first node (group creation or join). Further, the sequence of group operations is traversed one by one. If the sender is not in the group according to the current membership set, the operation is dropped. Else, the set is updated by adding or removing the specified member. The final membership set is the output.

At a very high level, [Fou23a] represents the history of a group as a collection of events, which are roughly the same thing as our group operations. In fact, [Fou23a] defines much more event types, but here we focus on those we have in common (it is not hard to extend our FR-CGKA by other types). Further, [Fou23a] distinguishes two types of events: control events that can change user's ability to send/receive, and state events that cannot. Since in this paper we only have control events, i.e., adds and removes ([Fou23a] does not have special update events), we ignore state events. Relations between control events are represented as a directed acyclic graph (DAG). An edge from event E

[8]In the propose-commit syntax of [BB$^+$20], these group changes can be done as a batch modification. Our syntax easily extends to batch operations.

[9]To make a connection with history graphs, we flip the arrow direction from [Fou23a].

Predicate P for FREEK

// P is true if the adversary leaked the (unpunctured) init secret from *any* id′ and exposed id's individual secrets

$$P(\mathsf{Eps}, \mathsf{id}, V, \mathsf{Move}^{\mathcal{A}}, *, \mathsf{Visited}^{\mathcal{A}}) \leftrightarrow \exists \mathsf{id}' \ \big(\mathsf{id}', \mathsf{Eps}[V].\mathsf{par}\big) \in \mathsf{Move}_i^{\mathcal{A}} \land (\mathsf{id}', V) \notin \mathsf{Visited}_i^{\mathcal{A}}\big) \land \ \texttt{*leaked-ind-secs}(\mathsf{id}, V)$$

$$\texttt{*leaked-ind-secs}(U, \mathsf{id}) \leftrightarrow \big(\exists U' : \texttt{*share-ind-secs}(U, U', \mathsf{id}) \land \texttt{*ind-secs-bad}(U', \mathsf{id})\big)\texttt{*exposed-ind-secs-weak}(U, \mathsf{id})$$

$\texttt{*share-ind-secs}(U, U', \mathsf{id}) \leftrightarrow U$ and U' are the same or connected via undirected path of epochs U'' such that $\mathsf{Eps}[U''].\mathsf{sndr} \neq \mathsf{id} \land \mathsf{Eps}[U].\mathsf{act} \notin \{\text{‘}rem\text{’-}\mathsf{id}, \text{‘}add\text{’-}\mathsf{id}\}$

$$\texttt{*ind-secs-bad}(U, \mathsf{id}) \leftrightarrow \boxed{(\mathsf{id}, U) \in \mathsf{Move}^{\mathcal{A}}} \lor (\mathsf{Eps}[U].\mathsf{packet} = \text{‘}inj\text{’} \land (\mathsf{Eps}[U].\mathsf{sndr} = \mathsf{id} \lor \mathsf{Eps}[U].\mathsf{act} = \text{‘}add\text{’-}\mathsf{id}))$$

$\texttt{*exposed-ind-secs-weak}(U, \mathsf{id}) \leftrightarrow \exists U_1, U_2, U_3$: all of the following conditions are satisfied:

(1) $U_1 \neq U_2 \land \texttt{*}\mathsf{ancestor}(U_1, U) \land \texttt{*}\mathsf{ancestor}(U_2, U_3)$ (3) $\texttt{*share-ind-secs}(U_1, U, \mathsf{id}) \land \texttt{*share-ind-secs}(U_2, U_3, \mathsf{id})$

(2) $\mathsf{Eps}[U_1].\mathsf{act} = \mathsf{Eps}[U_2].\mathsf{act} = \text{‘}add\text{’-}\mathsf{id}$ (4) $\mathsf{Eps}[U_2].\mathsf{inj} \land \mathsf{id} \in \mathsf{Eps}[U_3].\mathsf{exp}$

Figure 15: The predicate P instantiating the generic security predicates for FREEK.

to E' indicates that a party generating E' knew about E at the time (in the protocol of [Fou23a], this relation is made explicit in the packet).[10] Finally, each state event is associated with the control event directly preceding it (in the generator's view).

Choosing epoch on which to base the new current epoch. The choice of U in Step 3) impacts the cost of transitioning to the new current epoch for different members. Moreover, creating a long chain of modifications may result in destroying TreeKEM's distributed state and super-logarithmic communication cost. Unfortunately, there is no one-size-fits-all solution. For example, some applications may want to minimize the average cost for a group member to transition, while others — the cost for a couple of constrained users (e.g., IoT devices).

7 Security of FREEK

FREEK's security predicate. FREEK's security predicates **conf** and **auth** instantiate the generic security predicates from Fig. 9 with the predicate P defined in Fig. 15.

Recall that according to the optimal predicates ($P(\cdot) = \texttt{false}$), the adversary can put a move pebble for id on V only if it has a move pebble *for* id on V's parent U and there is no visited pebble *for* id on V. At a high level, the reason why FREEK achieves a weaker guarantee is that some secrets used by id (the init secret and the MAC key) are shared among all group members. Therefore, it is enough to have a move pebble on U without a visited pebble on V *for any* id′ to get these secrets. Getting these secrets may break the last line of defense and enable putting a move pebble for id on V.

A bit more precisely, we can distinguish two types of FREEK secrets for each epoch: *group secrets* known to all group members and *individual secrets* known only to some members (e.g., the signing key of id or PKE secret keys generated for it by TreeKEM). FREEK guarantees that the adversary can put a move pebble for id on V only if A) it has the group secrets of its parent U and B) id's individual secrets in V. For A), it is enough for the adversary to have a move pebble on U for any id′ (c.f. the first two literals of P). To decide if B) is satisfied, we use the predicate `*leaked-ind-secs` which is the same as in SAIK.

It remains to explain `*leaked-ind-secs`. It consists of two clauses, where the latter one considers an edge-case attack on SAIK; we refer to [AHKM22a] for an explanation and focus here on the former, more interesting case. Roughly, id's secrets are replaced by FREEK whenever it sends a message

[10] To make a connection with history graphs, we flip the arrow direction compared to [Fou23a].

by (possibly) secure ones.[11] Its secrets are also not present in epochs where it is not a member. We define `*share-ind-secs`(U, U', id) to capture when id's secrets are the same in U and U'. Now `*leaked-ind-secs` is true in U if they are the same as in some U' where they are corrupted, as per `*ind-secs-bad`. The latter is true if id's state is exposed in U', which is marked by $\mathcal{F}_{\text{FR-CGKA}}$ by adding a move pebble, or id's state is created via an injection on its behalf, or an injection that adds it.

FREEK vs SAIK. Since FR-CGKA implies CGKA, it makes sense to ask how security of SAIK compares with the security of FREEK when used in the CGKA mode, i.e., when every party has only one move pebble on the "current" epoch (when transitioning, the old move pebble is immediately deleted). It turns out that FREEK in the CGKA mode achieves the same CGKA security as SAIK. In fact, the protocols are almost equivalent, since puncturing becomes void — any keys punctured by FREEK when creating new epochs are immediately deleted.

We note that SAIK uses simplified predicates that do not enforce deleting group keys after outputting them. In our case, this means that key and move pebbles become the same thing and the former are not needed — an epoch is secure if the adversary does not have a move pebble on it. With this said, equivalence of SAIK's predicate and our predicate without key pebbles easily follows by inspection.

7.1 Security Proof of FREEK

In the current section we prove security of FREEK. Before presenting the full proof we provide the high level idea behind it. At a high level, we require the following properties from the underlying primitives. For the PPRF, we require one-wayness security (with adaptive corruptions), defined formally in Sec. 2.5. The TreeKEM component of FREEK, which is the same as in FREEK, also requires a multi-message multi-recipient PKE scheme, mmPKE, which we recall in Sec. 2. At a high level, mmPKE has the functionality (and security) of a parallel composition of standard PKEs. Its goal is to improve efficiency. For mmPKE, we require one-wayness (the adversary's goal is to compute a random encrypted message) RCCA (a relaxation of CCA) security. See Sec. 2.

Proof intuition. We prove our theorem using a sequence of hybrid experiments $\mathbf{Hyb}^i$ (transitioning from the real to the ideal execution). Hybrid $\mathbf{Hyb}^0$ is equal to the real world execution. Hybrid $\mathbf{Hyb}^1$ is identical to $\mathbf{Hyb}^0$, but translated to the ideal-world language. That is, we replace the real world execution with FREEK by an execution with a "trivial" ideal functionality $\mathcal{F}^1_{\text{FR-CGKA}}$ that forwards all inputs to $\mathcal{S}^1$ and returns the values set by it. $\mathcal{S}^1$ simply runs the code of FREEK on the inputs. The indistinguishability between $\mathbf{Hyb}^0$ and $\mathbf{Hyb}^1$ is straightforward.

Agreement and step correctness. The next hybrid, $\mathbf{Hyb}^2$, contains the functionality $\mathcal{F}^2_{\text{FR-CGKA}}$, which is the same as $\mathcal{F}_{\text{FR-CGKA}}$ except that it uses **conf** = **auth** = `false`. I.e., it does not enforce that the protocol provides authenticity or confidentiality, but it does enforce step correctness.

Intuitively, the main differences between the current hybrid and the previous one are the following: i) the outputs of the protocol are now set by the functionality $\mathcal{F}^2_{\text{FR-CGKA}}$ (instead of being directly set by the simulator as in the previous one), and ii) the execution can be disrupted if the checks made by $\mathcal{F}^2_{\text{FR-CGKA}}$ via **assert** statements and the helpers, fail.

First we deal with i). Regarding, calls to `*assert-agree-auth-preserved`, the execution is disrupted only when one of three assertions made by the helper function (cf. Fig. 8) fail. For the first

[11]This is the difference from the optimal protocol — there id's secrets are replaced with each epoch change, without interaction with id.

and third assertion, the claim follows by inspection. For the second assertion, recall that V is equal to $\mathsf{HKDF.Exp}(\gamma.\mathsf{grpCtxt}(), \mathsf{comSecConf})$, thus a cycle is created if for two different epochs (with different parents and thus different group contexts), the output of $\mathsf{HKDF.Exp}$ is the same, which implies a collision against $\mathsf{HKDF.Exp}$.

Regarding, `*step-correct`, we show that in the current hybrid, if when an id receives a message `*step-correct` is true, then the protocol run by $\mathcal{S}^2$ accepts the input. Observe that id's protocol rejects such a packet only if A) id does not have the epoch U in its state, B) it possesses the state information for the epoch U, but the init secret $\mathsf{initParSec}_U$ has been already punctured on the epoch id of the target epoch, V, or C) the correctness property of the signature or the encryption scheme fails, i.e., for an honestly generated packet, signature verification or decryption, fails. It is not hard to see that A), B) would contradict `*step-correct`, while C) violates the (perfect) correctness of the either the encryption, or the signature, scheme. Finally, *indistinguishability of outputs*, (case i)), follows by inspection and the collision resistance of $\mathsf{HKDF.Exp}$ (cf. Sec. 7.1).

Confidentiality. The next hybrid, $\mathbf{Hyb}^3$, introduces confidentiality, which is formalized by restoring the original **conf** predicate of $\mathcal{F}_{\text{FR-CGKA}}$. The simulator $\mathcal{S}^3$ is the same as $\mathcal{S}^2$. Assuming security of PKE and PPRF, and RO, we show indistinguishability between the current hybrid and the previous one. Observe that the only difference between $\mathbf{Hyb}^2$ and $\mathbf{Hyb}^3$ is that group keys in confidential epochs are real in hybrid $\mathbf{Hyb}^2$ (technically, computed by the simulator according to FREEK) and random and independent in $\mathbf{Hyb}^3$ (technically, sampled by $\mathcal{F}^3_{\text{FR-CGKA}}$). Since application secrets (i.e., group keys) are derived by hashing the respective epoch secrets (recall that $\mathsf{appSec} \leftarrow \mathsf{HKDF.Exp}(\mathsf{epSec}, \textit{'app'})$ and appSec is I in $\mathcal{F}^3_{\text{FR-CGKA}}$), the distinguishing advantage of $\mathcal{A}$ is upper-bounded by the probability that it inputs to the RO the epoch secret, epSec, from some confidential epoch V. Therefore, we focus on upper-bounding the probability of the aforementioned event. Since the epoch secret is the HKDF.Exp (i.e., RO) output over the joiner secret, where the latter depends on other secrets, in the full proof, we define a sequence of bad events, which are related to queries of other secrets. Let $\mathsf{Q}_{\mathcal{A}}(X)$ be the event that $\mathcal{A}$ queries X to the RO and V_0 be the first epoch. In the full proof, after a sequence of calculations over predicates and events we conclude that the adversary manages to distinguish between the two hybrids, if:
a) Exists s in $\{\mathsf{epSec}_{V_0}, \mathsf{joinerSec}_{V_0}, \mathsf{initParSec}_{V_0}\}$ s.t. $\mathsf{Q}_{\mathcal{A}}(s)$ and $\mathbf{conf}(V)$,
b) Or exists V with parent U s.t. for $s \in \{\mathsf{epSec}_U, \mathsf{joinerSec}_U, \mathsf{initParSec}_U\}$, $\neg\mathsf{Q}_{\mathcal{A}}(s)$, V is not reachable (see below) from U and $\mathsf{Q}_{\mathcal{A}}(\mathsf{initChildSec}_{U,V})$,
c) Or exists V s.t. $\mathsf{Q}_{\mathcal{A}}(\mathsf{commitSec}_V)$ and $\mathbf{conf}(V)$,
d) Or exists V s.t. $\mathbf{conf}(V)$, $\mathsf{Q}_{\mathcal{A}}(\mathsf{joinerSec}_V)$ and $\neg\mathsf{Q}_{\mathcal{A}}(\mathsf{initSec}(U,V), \mathsf{commitSec}_V)$,
e) Or exists V s.t. for some secret $s \neq \mathsf{joinerSec}_V$, $\mathsf{Q}_{\mathcal{A}}(s)$.

Cases a) and e) are bounded via straightforward probabilistic arguments related to the RO queries, since the view of $\mathcal{A}$ is independent of any protocol secrets. Cases c), d), rely on the fact that V is confidential, thus security is reduced to the security of the underlying PKE scheme. For this part we use existing lemmas from SAIK [AHKM22b]. It only remains to bound the probability of b). Observe that in b) $\mathcal{A}$ queries the output of PPRF, $\mathsf{initChildSec}_{U,V}$, to the RO, in a setting where it never queries the secrets of the parent epoch, and V is not reachable from U, meaning that 1) either $\mathcal{A}$ does not own any move pebbles w.r.t. U, or 2) the move pebbles where put after corrupting an id that had already visited V. 1) implies that the PPRF key of U is not known to the adversary, while 2) implies that $\mathcal{A}$ learns U's PPRF key after the key is punctured on V, therefore we bound the probability of b) via a reduction to the one-wayness property of PPRF (cf. Sec. 2.5): assuming $\mathcal{A}$ queries $\mathsf{initChildSec}_{U,V}$ to the RO we construct a reduction $\mathcal{B}$ that uses $\mathsf{initChildSec}_{U,V}$ and V to break one-waynes of PPRF.

Authenticity. The fourth and final hybrid introduces authenticity, which is formalized by restoring the **auth** predicate. This hybrid matches the ideal experiment with $\mathcal{F}_{\text{FR-CGKA}}$. In Sec. 7.1 we prove that, if Sig and MAC are unforgeable and if mmPKE is $\mathsf{mmOW\text{-}RCCA}$ secure, then FREEK guarantees authenticity, that is, the current hybrid is indistinguishable from the previous one.

Our result is formally presented in the following theorem. Let $\mathcal{F}_{\text{FR-CGKA}}$ be the functionality from Fig. 8 with predicates **conf**, **auth** defined in Fig. 9 and P defined in Fig. 15. Let mmPKE, PPRF, Sig, MAC and HKDF be the schemes instantiating FREEK. Denote the output of an environment $\mathcal{A}$ in the real execution with FREEK and the hybrid functionality $\mathcal{F}_{\text{AKS}}$ from Fig. 17 as $\text{REAL}_{\mathsf{FREEK},\mathcal{F}_{\text{AKS}}}(\mathcal{A})$ and the output of $\mathcal{A}$ in the ideal execution with $\mathcal{F}_{\text{FR-CGKA}}$ and a simulator $\mathcal{S}$ as $\text{IDEAL}_{\mathcal{F}_{\text{FR-CGKA}},\mathcal{S}}(\mathcal{A})$. For any $\mathcal{A}$, there exists an $\mathcal{S}$ and adversaries $\mathcal{B}_1$ to $\mathcal{B}_5$ s.t.

$$\begin{aligned}
&\Pr[\text{IDEAL}_{\mathcal{F}_{\text{FR-CGKA}},\mathcal{S}}(\mathcal{A}) = 1] - \Pr\left[\text{REAL}_{\mathsf{FREEK},\mathcal{F}_{\text{AKS}}}(\mathcal{A}) = 1\right] \leq \\
&\qquad \text{Adv}^{\mathsf{CR}}_{\mathsf{HKDF.Exp}}(\mathcal{B}_1) + 2 \cdot \Big(1/2^{\kappa} + \text{Adv}^{\mathsf{OW\text{-}PPRF}}_{\mathsf{PPRF},q_e,q_m}(\mathcal{B}_1) \\
&\qquad + q_e \cdot \text{Adv}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE},q_e \log(q_n),q_n}(\mathcal{B}_2) + q_e \cdot \text{Adv}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE},1,q_n}(\mathcal{B}_3) + \frac{s \cdot q_e \cdot q_h}{2^{\kappa}}\Big), \\
&\qquad + 2q_e \cdot \text{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig}}(\mathcal{B}_3) + \ q_e \cdot \text{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{B}_4),
\end{aligned}$$

where q_e, q_n, q_m, and q_h, denote bounds on the number of epochs, the group size, the number of children of a single epoch, and the number of $\mathcal{A}$'s queries to the random oracle modeling HKDF, respectively, and $s = 5$ denotes the number of key schedule secrets in a single epoch.

Proof. We prove our theorem using a sequence of hybrid experiments $\mathbf{Hyb}^i$ (transitioning from the real to the ideal execution) and we will use subscripts as with IDEAL to parameterize our hybrids with functionalities and simulators.

$\mathbf{Hyb}^0$: This hybrid is equal to the real world execution, $\text{REAL}_{\mathsf{FREEK},\mathcal{F}_{\text{AKS}}}(\mathcal{A})$.

$\mathbf{Hyb}^1_{\mathcal{F}^1_{\text{FR-CGKA}},\mathcal{S}^1}$: The current hybrid is identical to $\mathbf{Hyb}^0$, but translated to the ideal-world language. That is, we replace the real world execution with FREEK and $\mathcal{F}_{\text{AKS}}$ by an execution with a "trivial" ideal functionality $\mathcal{F}^1_{\text{FR-CGKA}}$ and a "trivial" simulator $\mathcal{S}^1$. $\mathcal{F}^1_{\text{FR-CGKA}}$ gives no security guarantees but matches the syntax of $\mathcal{F}_{\text{FR-CGKA}}$. That is, $\mathcal{F}^1_{\text{FR-CGKA}}$ forwards all inputs to $\mathcal{S}^1$ and allows it to choose all outputs (protocol messages, group keys, etc). $\mathcal{S}^1$ uses these inputs to run the code of FREEK and $\mathcal{F}_{\text{KS}}$, which allows it to compute the outputs exactly as in the real world.

Lemma 1. *The view of $\mathcal{A}$ in $\mathbf{Hyb}^0$ is identical to the one in $\mathbf{Hyb}^1_{\mathcal{F}^1_{\text{FR-CGKA}},\mathcal{S}^1}$.*

Proof. Straightforward by inspection. □ □

In the subsequent hybrids, we modify $\mathcal{F}^1_{\text{FR-CGKA}}$, which gives no security guarantees, by gradually introducing the security guarantees of FR-CGKA. We do so by gradually activating the security predicates and the checks (made by our helper functions via assertions), that enforce the properties of our protocol, i.e., consistency, step correctness, confidentiality and authenticity. In particular, $\mathcal{F}^2_{\text{FR-CGKA}}$ introduces consistency and step correctness, $\mathcal{F}^3_{\text{FR-CGKA}}$ introduces confidentiality, and $\mathcal{F}^4_{\text{FR-CGKA}}$ authenticity. The simulator is modified accordingly, to deal with less power given by the functionality. We first deal with consistency and correctness.

7.2 Consistency and Step Correctness

We show that FREEK guarantees step correctness and consistency, by showing indistinguishability between $\mathbf{Hyb}^1_{\mathcal{F}^1_{\text{FR-CGKA}},\mathcal{S}^1}$ and the second hybrid that we define below w.r.t. a new simulator $\mathcal{S}^2$.

$\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}$: The current hybrid contains the functionality $\mathcal{F}^2_{\text{FR-CGKA}}$, which is the same as $\mathcal{F}_{\text{FR-CGKA}}$ except that it uses **conf** = **auth** = `false`. I.e., it disables all checks related to authenticity and confidentiality of the protocol while consistency and step correctness checks are enabled (by default). In a high level, the current hybrid is distinguishable from the previous one (which is identical to real world execution), if the environment can violate consistency of the execution or step correctness. The simulator $\mathcal{S}^2$ is defined as follows:

Simulator $\mathcal{S}^2$

$\mathcal{S}^2$ stores a FREEK state for each party id. When $\mathcal{F}^2_{\text{FR-CGKA}}$ sends to $\mathcal{S}^2$ an FR-CGKA input from id (for `Send`, `Receive`, `Join`, `GetKey`, or `DeleteMovePebble`), $\mathcal{S}^2$ runs FREEK on this input and updates the state. It sends to $\mathcal{F}^2_{\text{FR-CGKA}}$ the epoch identifier(s) U, V and well as any packets outputted by the protocol. For injected epochs, it sends to $\mathcal{F}^2_{\text{FR-CGKA}}$ the values sndr, act and mem taken from the ratchet tree in id's state. When id gets corrupted, $\mathcal{S}^2$ sends its state to the environment $\mathcal{A}$.

Indistinguishability between the current hybrid and the previous one is proven via the following lemma.

Lemma 2. *For any environment $\mathcal{A}$, there exists an adversary $\mathcal{B}$ such that*

$$\Pr\left[\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}(\mathcal{A}) \Rightarrow 1\right] - \Pr\left[\mathbf{Hyb}^1_{\mathcal{F}^1_{\text{FR-CGKA}},\mathcal{S}^1}(\mathcal{A}) \Rightarrow 1\right] \leq \mathrm{Adv}^{\mathsf{CR}}_{\mathsf{HKDF.Exp}}(\mathcal{B}).$$

The probability runs over the randomness used by the experiments.

Proof. The main differences between the current hybrid ($\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}$) and the previous one ($\mathbf{Hyb}^1_{\mathcal{F}^1_{\text{FR-CGKA}},\mathcal{S}^1}$), are the following: (1) the outputs of the protocol are now set by the functionality $\mathcal{F}^2_{\text{FR-CGKA}}$ (instead of being directly set by the simulator as in $\mathbf{Hyb}^1_{\mathcal{F}^1_{\text{FR-CGKA}},\mathcal{S}^1}$), and (2) the execution can be disrupted if the checks made by $\mathcal{F}^2_{\text{FR-CGKA}}$ via **assert** statements and the helpers, fail. First we deal with (2).

Helper `*assert-agree-auth-preserved`. We show that calls to `*assert-agree-auth-preserved` do not disrupt the execution, which only happens if one of the following three assertions fail:

1. **assert** $\forall U$ s.t. $\mathsf{Eps}[U] \neq \perp \ : \ \{\mathsf{id} \ : \ (\mathsf{id}, U) \in \mathsf{Visited}\} \subseteq \mathsf{Eps}[U].\mathsf{mem}$
2. **assert** HG has no cycles
3. **assert** $\forall U$ s.t. $\mathsf{Eps}[U] \neq \perp \wedge \mathsf{Eps}[U].\mathsf{par} \neq \perp \ : \mathsf{Eps}[U].\mathsf{mem} = \texttt{*mem}(\mathsf{Eps}[U].\mathsf{par}, \mathsf{Eps}[U].\mathsf{act})$

For case 1, observe that (id, U) can be added to Visited via `Initialization`, `Send`, `Receive` or `Join`. We will show that such id also belongs to $\mathsf{Eps}[U].\mathsf{mem}$. For `Initialization`, id instantly becomes a member of the new epoch via a call to `*new-ep` with input $\mathsf{mem} = \{\mathsf{id}\}$. For `Send`, id becomes a member of V via a call to $\texttt{*mem}(U, \mathsf{act})$, which copies all group members of U. For `Receive`, if $\mathsf{Eps}[V] \neq \perp$, the receiver has been included to the set of group members of V when V was created, otherwise, it becomes a member of V by calling `*new-ep` (which copies the group members). Finally, for `Join`, if $\mathsf{Eps}[U] \neq \perp$, id has already become a group member by the `Send` operation that issued the add of id, via a call to `*new-ep` with input $\mathsf{mem} = \texttt{*mem}(U, \mathsf{act})$, otherwise, membership is set by the simulator as in the previous hybrid. Using similar arguments and the fact that the protocol handles removals consistently (a `Send` operation that removes id creates a new epoch in which id is removed), we observe that the assertion of case 3 also holds.

For case 2, recall that, by the definition of the key schedule, the epoch id of a newly generated epoch is $V = \mathsf{HKDF.Exp}(\gamma.\mathsf{grpCtxt}(), \mathsf{comSecConf})$, where $\gamma.\mathsf{grpCtxt}()$ includes the epoch id of the parent epoch. Assume $\mathcal{A}$ creates a cycle for the first time. This implies that there exist $U \neq V$, such that

$$\begin{aligned}\mathsf{HKDF.Exp}(\gamma.\mathsf{grpCtxt}_V(), \mathsf{comSecConf}_V) = V& \\ = \mathsf{HKDF.Exp}(\gamma.\mathsf{grpCtxt}_U(), \mathsf{comSecConf}_U).&\end{aligned}$$

Since $U \neq V$, the parent ids in $\gamma.\mathsf{grpCtxt}_U()$, $\gamma.\mathsf{grpCtxt}_V()$, are different, therefore, $\gamma.\mathsf{grpCtxt}_U() \neq \gamma.\mathsf{grpCtxt}_V()$, and we can define an adversary $\mathcal{B}$ that simulates the execution with $\mathcal{A}$ and outputs a collision against $\mathsf{HKDF.Exp}$, reaching a contradiction. Therefore, there can be no cycles in the history graph.[12]

Helper `*step-correct`. We show that in $\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}}, \mathcal{S}^2}$, if `*step-correct` is true when an id receives a message, then the protocol executed by $\mathcal{S}^2$ accepts the input. That is, the input refers to a packet that enables the transition from U to V, correctly extracted by $\mathcal{F}^2_{\text{FR-CGKA}}$. Notice that id's protocol rejects such a packet only if A) id does not have the epoch U in its state, B) it possesses the state information for the epoch U, but the init secret $\mathsf{initParSec}_U$ has been already punctured on the epoch id of the target epoch, V, or C) the correctness property of the signature or the encryption scheme fails, i.e., for an honestly generated packet, signature verification or decryption, fails.

Regarding A), it only happens if id has never transitioned to U, or it executed DeleteMovePebble w.r.t. U. This implies that $(\mathsf{id}, U) \notin \mathsf{Move}$, which contradicts `*step-correct`. Furthermore, regarding B), if $\mathsf{initParSec}_U$ is already punctured on V, given the fact that there are no cycles (as shown above), this means that id has already visited V and there cannot be another epoch with the same identifier, which again contradicts `*step-correct`. Finally, C) violates the (perfect) correctness of the either the encryption, or the signature, scheme.

Indistinguishability of outputs. Now that we have proved that the checks introduced by $\mathcal{F}^2_{\text{FR-CGKA}}$ do not affect indistinguishability, we show that the parties' outputs observed by $\mathcal{A}$ are the same in both hybrids. In $\mathbf{Hyb}^1_{\mathcal{F}^1_{\text{FR-CGKA}}, \mathcal{S}^1}$, the outputs are directly forwarded from $\mathcal{S}^1$ executing FREEK. In $\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}}, \mathcal{S}^2}$, they are computed by $\mathcal{F}^2_{\text{FR-CGKA}}$ with the help of the values sent by $\mathcal{S}^2$. Let us observe all possible outputs. First, the epoch identifiers U, V, as well as the packet C, are chosen by the simulator, so they are identical in the two hybrids. Thus, it only remains to show that sndr and act, output by `Receive`, sndr and mem, output by `Join`, and the key, key, output by `GetKey`, are indistinguishable from those output by the protocol (and forwarded by $\mathcal{S}^1$) in hybrid $\mathbf{Hyb}^1_{\mathcal{F}^1_{\text{FR-CGKA}}, \mathcal{S}^1}$. Observe that for those operations, if the "if" statements are not satisfied (which implies that the target epoch is generated on-the-fly by the simulator as the operation executes), the execution follows a path in which all values are set by the simulator, thus for those cases the outputs match those of the previous hybrid. If the "if" statements are satisfied (which implies that the target epoch has been already created by a previous operation), then we need to make sure that the properties of the epoch (namely, sndr, act, mem, key) at the time of creation, match the ones returned to the parties in

[12]Here, we assume that the id of the root epoch is not in the range of the $\mathsf{HKDF.Exp}$. This trivially excludes collisions against the root epoch.

subsequent calls, i.e., the parties' CGKA states are consistent. In order to prove that (and conclude the proof of Lemma 2) we rely on the following claim.

Claim 1. *In* $\mathbf{Hyb}^1_{\mathcal{F}^1_{\text{FR-CGKA}},\mathcal{S}^1}$, $\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}$, *if each two parties* id_1 *and* id_2*, with states* γ_{id_1}, γ_{id_2}*, respectively, (emulated by* $\mathcal{S}^1$, $\mathcal{S}^2$*, respectively), transition to a new epoch (after a* `Send`*,* `Receive` *or* `Join`*, operation) and they both output the same epoch identifier,* V*, then either there is an* $\mathsf{HKDF.Exp}$ *collision or*

1. $\gamma_{\mathsf{id}_1}[V].\tau = \gamma_{\mathsf{id}_2}[V].\tau$ *and*[13]
2. $\gamma_{\mathsf{id}_1}[V].\mathsf{keySchedule} = \gamma_{\mathsf{id}_2}[V].\mathsf{keySchedule}$ *and*
3. $\gamma_{\mathsf{id}_1}[V].\mathsf{parent} = \gamma_{\mathsf{id}_2}[V].\mathsf{parent}$ *and*
4. *The last group modification,* act*, outputted by* id_1 *and* id_2 *is the same (if one of them joins, the implicit modification is adding them).*

With the above claim observe that, if the epoch identifier matches, then since $V = \mathsf{HKDF.Exp}(\gamma.\mathsf{grpCtxt}_V(), \mathsf{comSecConf}_V)$ and $\gamma.\mathsf{grpCtxt}_V$ contains the last action and sender, sndr and act are the same for all parties that reach V (for all operations) assuming collision resistance of $\mathsf{HKDF.Exp}$. $\gamma.\mathsf{grpCtxt}_V$ also includes the tree hash, thus the member set outputted on Join is consistent with the tree in the states of other parties transitioning to the epoch. By the fact that the last action is the same and recursion, this is consistent with the member set in the functionality. It remains to prove that the key output by `GetKey` is consistent. By collision resistance, all parties that reach V agree on $\gamma.\mathsf{grpCtxt}_V()$ and $\mathsf{comSecConf}_V$, which implies they also agree on $\mathsf{pathSec}_V$, since $\mathsf{comSecConf}_V = \mathsf{HKDF.Exp}(\mathsf{pathSec}_V, \textit{'conf'})$ (by collision resistance), therefore they all compute the same $\mathsf{commitSec}_V = \mathsf{HKDF.Exp}(\mathsf{pathSec}_V, \textit{'com'})$. Finally, $\gamma.\mathsf{grpCtxt}_V$ contains the id of the parent epoch, thus $\mathsf{initParSec}_U$ and $\mathsf{initChildSec}_{U,V} = \mathsf{PPRF.eval}(\mathsf{initParSec}_U, V)$ should be the same for all parties that reach V. We conclude that $\mathsf{joinerSec}_V$ is the same for all parties and thus the group key is also the same. Clearly, if any of the above is violated we can construct an adversary $\mathcal{B}$ that uses $\mathcal{A}$ to break collisions resistance of $\mathsf{HKDF.Exp}$. The proof of the above claim follows.

of the above claim. The proof follows directly from the way the epoch identifier V and the key schedule are computed using $\mathsf{HKDF.Exp}$. We revisit the relevant parts of FREEK below.

- The group context, $\gamma.\mathsf{grpCtxt}_V()$, of V contains (the tree hash of) the ratchet tree in epoch V, the identifier U of the parent of V and the last group modification.
- The path secret $\mathsf{pathSec}_V$ is chosen by the party creating V.
- The identifier of the new epoch is $V = \mathsf{HKDF.Exp}(\gamma.\mathsf{grpCtxt}_V(), \mathsf{comSecConf}_V)$, where $\mathsf{comSecConf}_V = \mathsf{HKDF.Exp}(\mathsf{pathSec}_V, \textit{'conf'})$
- The key schedule of V is derived from $\mathsf{joinerSec}_V$ computed as the $\mathsf{HKDF.Ext}$ of $\gamma.\mathsf{grpCtxt}_V()$, $\mathsf{initChildSec}_{U,V} = \mathsf{PPRF.eval}(\mathsf{initParSec}_U, V)$ and $\mathsf{commitSec}_V = \mathsf{HKDF.Exp}(\mathsf{pathSec}_V, \textit{'comm'})$.

The proof of the claim is concluded. This concludes the proof of Lemma 2.

$\square$

$\square$

[13]Here, we refer only to the public part of the CGKA states of the users in epoch V.

7.3 Confidentiality

The next hybrid introduces confidentiality, which is formalized by restoring the original confidentiality predicate of $\mathcal{F}_{\text{FR-CGKA}}$.

$\mathbf{Hyb}^3_{\mathcal{F}^3_{\text{FR-CGKA}},\mathcal{S}^3}$: The functionality $\mathcal{F}^3_{\text{FR-CGKA}}$ uses the original **conf** predicate from $\mathcal{F}_{\text{FR-CGKA}}$. The simulator $\mathcal{S}^3$ is the same as $\mathcal{S}^2$.

We next show that if the PKE and PPRF schemes are secure, then $\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}$ and $\mathbf{Hyb}^3_{\mathcal{F}^3_{\text{FR-CGKA}},\mathcal{S}^3}$ are indistinguishable for any PPT environment $\mathcal{A}$.

Proof intuition.

For better intuition, observe that hybrids $\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}$ and $\mathbf{Hyb}^3_{\mathcal{F}^3_{\text{FR-CGKA}},\mathcal{S}^3}$ are almost identical. In both experiments, the environment interacts with the CGKA functionality and the same simulator. The only difference is that group keys in confidential epochs are real in hybrid $\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}$ (technically, computed by the simulator according to FREEK) and random and independent in $\mathbf{Hyb}^3_{\mathcal{F}^3_{\text{FR-CGKA}},\mathcal{S}^3}$ (technically, sampled by $\mathcal{F}^3_{\text{FR-CGKA}}$). This means that distinguishing between the two hybrids an be seen as a typical confidentiality game for CGKA schemes. The adversary in the game corresponds to the environment $\mathcal{A}$. The adversary's challenge queries correspond to $\mathcal{A}$'s GetKey inputs on behalf of parties in confidential epochs and its reveal-session key queries correspond to $\mathcal{A}$'s GetKey inputs in non-confidential epochs. To disable trivial wins, confidential epochs where a random key has been outputted are marked by setting a flag chall. $\mathcal{A}$ and the adversary in the game are not allowed to corrupt if this makes such an epoch non-confidential.

The indistinguishability between the two hybrids if formalized in the following lemma:

Lemma 3. *For any environment $\mathcal{A}$, there exist adversaries $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$, such that*

$$\begin{aligned}\Pr\left[\mathbf{Hyb}^3_{\mathcal{F}^3_{\text{FR-CGKA}},\mathcal{S}^3}(\mathcal{A}) \Rightarrow 1\right] - \Pr\left[\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}(\mathcal{A}) \Rightarrow 1\right] \leq&\\ 1/2^\kappa + \mathrm{Adv}^{\mathsf{OW\text{-}PPRF}}_{\mathsf{PPRF},q_e,q_m}(\mathcal{B}_1) +&\\ q_e \cdot \mathrm{Adv}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE},q_e \log(q_n),q_n}(\mathcal{B}_2) +&\\ q_e \cdot \mathrm{Adv}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE},1,q_n}(\mathcal{B}_3) + \frac{s \cdot q_e \cdot q_h}{2^\kappa}&,\end{aligned}$$

where HKDF.Exp *and* HKDF.Ext *are modeled as random oracles and q_e, q_m, q_n and q_h, are as in the main theorem. The probability runs over the randomness used by the experiments and the random oracle.*

Proof. First we define a predicate and events that will assist our proof.

The original **conf** predicate states that the environment cannot reach a configuration in which the set of it's keys contains U, i.e., adv can compute the key of epoch U.

Simpler predicate.

For simplicity, we first consider a slightly weaker confidentiality predicate : $\mathbf{conf}'(U)$ is true if

$$\nexists(\mathsf{Move}^{\mathcal{A}}_n, \_, \_) : \ (\mathsf{Move}^{\mathcal{A}}, \mathsf{Key}^{\mathcal{A}}, \mathsf{Visited}^{\mathcal{A}}) \vdash^* (\mathsf{Move}^{\mathcal{A}}_n, \_, \_) \wedge \exists \mathsf{id}(\mathsf{id}, U) \in \mathsf{Move}^{\mathcal{A}}_n.$$

We observe that $\mathbf{conf}'$ implies $\mathbf{conf}$: if $\mathbf{conf} = \mathtt{false}$, then by definition, $\mathcal{A}$ can reach a configuration in which it can compute they key of U, but this wouldn't be possible without knowing the state of U, which implies a move pebble on U, thus $\mathbf{conf}' = \mathtt{false}$. Furthermore, the only case for which

conf does not imply **conf′** (w.r.t. epoch U), i.e., the case in which the environment can't compute the key of U but there is a move pebble on U, are those where a) all parties have been corrupted after computing/deleting the key of U (via `GetKey`) and b) the environment cannot derive the key indirectly. Security of FREEK in these epochs is trivial, therefore for the rest of the proof we prove indistinguishability between the current hybrid and previous one by considering **conf′** in the place of **conf**.

Events.

Let $\mathcal{A}$ be any environment. Observe that, since application secrets are derived by hashing the respective epoch secrets (recall that $\mathsf{appSec} \leftarrow \mathsf{HKDF.Exp}(\mathsf{epSec}, \textit{`app'})$ and appSec is I in $\mathcal{F}^3_{\text{FR-CGKA}}$), the distinguishing advantage of $\mathcal{A}$ is upper-bounded by the probability that it inputs to the RO the epoch secret, epSec, from some confidential epoch V. Therefore, it remains to upper-bound the probability of $\mathcal{A}$ inputting the epoch secret of such a V to the RO.

We next define a number of predicates, each taking as an implicit input the execution (at some point in time) of the experiment with $\mathcal{A}$ and as an explicit input one or two epochs within it. One can think of the predicates, as well as formulas defined on them, as events occurring whenever a predicate becomes true. Note that, whenever we say $\mathcal{A}$ queries the RO with a secret, we assume that the whole input (to the RO) matches the format that the protocols uses.

// Eventually, the goal is to upper-bound the probability that there exists an V for which the next two predicates are true.

- $\mathsf{conf}(V)$ **:** $\mathbf{conf}'(V)$ is true.
- $\mathsf{hashesSec}(V)$ **:** An input of $\mathcal{A}$ to RO contains epSec_V, $\mathsf{joinerSec}_V$ or $\mathsf{initParSec}_V$.

// The next two predicates describe one way $\mathcal{A}$ can make $\mathsf{hashesSec}$ true : by hashing the init and commit to get the joiner

- $\mathsf{hashesIni}(U, V)$ **:** V is a child of U and some input of $\mathcal{A}$ to the RO contains $\mathsf{initChildSec}_{U,V}$ (the child init secret derived for V from $\mathsf{initParSec}_U$ via PPRF).
- $\mathsf{hashesCom}(V)$ **:** An input of $\mathcal{A}$ to RO contains $\mathsf{commitSec}_V$.

// Another way $\mathcal{A}$ can make $\mathsf{hashesSec}$ true is to get the joiner from other sources e.g. a welcome message.

- $\mathsf{hashesJoiOnly}(V)$ **:** An input of $\mathcal{A}$ to the RO contains $\mathsf{joinerSec}_V$ but no input contains $\mathsf{initChildSec}(U, V)$ and $\mathsf{commitSec}_V$ together.

// Finally, $\mathcal{A}$ could guess some secret derived from joiner without hashing the joiner.

- $\mathsf{guesses}(V)$ **:** An input of $\mathcal{A}$ to the RO contains some secret in V but no input to the RO contains $\mathsf{joinerSec}_V$.

The event that $\mathcal{A}$ can win w.r.t. an epoch V.

Let us look at any epoch V with parent U. The goal is to bound the probability of $\mathcal{A}$ winning w.r.t. V, which is captured by $\mathsf{hashesSec}(V) \wedge \mathsf{conf}(V)$. Observe that this event is slightly more general, as besides the queries made to the RO for epSec_V, it also considers queries of $\mathsf{joinerSec}_V$ and $\mathsf{initParSec}_V$.

We include $\mathsf{joinerSec}_V$ and $\mathsf{initParSec}_V$, in $\mathsf{hashesSec}(V)$ since the security of V relies on the security of prior epochs and $\mathsf{hashesSec}(V)$ will enable us to prove security of V via a recursive argument.

Next we define a super-event implied by $\mathsf{hashesSec}(V) \wedge \mathsf{conf}(V)$. The super-event will be a conjunction of simpler events, the probability of which we will upper-bound separately for modularity; the final upper bound will follow by the union bound.

First, recall that the epoch secret, epSec_V, is the hash of the joiner secret, $\mathsf{joinerSec}_V$, and the joiner secret is the hash of the commit secret, $\mathsf{commitSec}_V$, with $\mathsf{initChildSec}_{U,V}$ (and the group context, $\mathsf{grpCtxt}()$, which is public). Therefore,

$$\begin{aligned} \mathsf{hashesSec}(V) \implies & (\mathsf{hashesIni}(U,V) \wedge \mathsf{hashesCom}(V)) \\ & \vee \mathsf{hashesJoiOnly}(V) \vee \mathsf{guesses}(V). \end{aligned} \tag{1}$$

If the left hand side of the above formula is true due to the query of epSec_V, then the right hand side is also true, as explained above. If the left hand side of the formula is true due to the query of $\mathsf{initParSec}_V$, then since this secret is the hash of epSec_V, the environment either queries epSec_V or simply guesses $\mathsf{initParSec}_V$, and the right hand side is also true. Finally, if the left hand side is true due to the query of $\mathsf{joinerSec}_V$, then either the environment queries $\mathsf{initChildSec}(U,V)$ and $\mathsf{commitSec}_V$, together, or it should only query $\mathsf{joinerSec}_V$. In both cases the right hand side of the formula becomes true.

Now, observe that $\mathbf{conf}'(V)$ does not depend on $\mathsf{Key}^{\mathcal{A}}$ and, given rules b) and c) of the pebbling-step validity, $\mathbf{conf}'(V)$ implies one of the following:

// $\mathcal{A}$ cannot reach a configuration from which it could put a move pebble on U for some id and id did not visit V

1. $\mathsf{cantReach}(U,V) \iff \nexists \mathsf{Config}^{\mathcal{A}}_{n'} : \mathsf{Config}^{\mathcal{A}} \vdash^* \mathsf{Config}^{\mathcal{A}}_{n'} \wedge \exists \mathsf{id}(\mathsf{id},U) \in \mathsf{Config}^{\mathcal{A}}_{n'}.\mathsf{Move}^{\mathcal{A}} \wedge (\mathsf{id},V) \notin \mathsf{Config}^{\mathcal{A}}_{n'}.\mathsf{Visited}^{\mathcal{A}}$

// $\mathcal{A}$ cannot make a pebbling step

2. $\nexists \mathsf{id} : \texttt{*leaked-ind-secs}(\mathsf{id},V)$.

From the above and Eq. (1) we derive the following:

$$\begin{aligned} \mathsf{hashesSec}(V) \wedge \mathsf{conf}(V) \implies & (\mathsf{hashesIni}(U,V) \wedge \mathsf{hashesCom}(V) \wedge \mathsf{cantReach}(U,V)) \\ & \vee (\mathsf{hashesIni}(U,V) \wedge \mathsf{hashesCom}(V) \wedge \\ & \qquad \nexists \mathsf{id} : \texttt{*leaked-ind-secs}(\mathsf{id},V))) \\ & \vee (\mathsf{hashesJoiOnly}(V) \wedge \mathsf{conf}(V)) \\ & \vee (\mathsf{guesses}(V) \wedge \mathsf{conf}(V)), \end{aligned}$$

which we can simplify as follows:

$$\begin{aligned} \mathsf{hashesSec}(V) \wedge \mathsf{conf}(V) \implies & (\mathsf{cantReach}(U,V) \wedge \mathsf{hashesIni}(U,V)) \\ & \vee (\mathsf{hashesCom}(V) \wedge \nexists \mathsf{id} : \texttt{*leaked-ind-secs}(\mathsf{id},V))) \\ & \vee (\mathsf{hashesJoiOnly}(V) \wedge \mathsf{conf}(V)) \\ & \vee \mathsf{guesses}(V). \end{aligned}$$

We now apply induction to the above formula and as a logical consequence we derive the following:

$$\begin{aligned}\mathsf{hashesSec}(V) \wedge \mathsf{conf}(V) \implies & (\mathsf{hashesSec}(U) \wedge \mathsf{conf}(U)) \\ & \vee (\neg\mathsf{hashesSec}(U) \wedge \mathsf{cantReach}(U,V) \wedge \mathsf{hashesIni}(U,V)) \\ & \vee (\mathsf{hashesCom}(V) \wedge \nexists \mathsf{id} : \texttt{*leaked-ind-secs}(\mathsf{id}, V))) \\ & \vee (\mathsf{hashesJoiOnly}(V) \wedge \mathsf{conf}(V)) \\ & \vee \mathsf{guesses}(V).\end{aligned}$$

Finally, denote the ancestors of V by $V_0, \dots, V_{\ell-1} = U, V_\ell = V$. By induction,

$$\begin{aligned}\mathsf{hashesSec}(V_\ell) \wedge \mathsf{conf}(V_\ell) \implies & (\mathsf{hashesSec}(V_0) \wedge \mathsf{conf}(V_0)) \\ & \vee (\exists i \in [1,\ell]\ \neg\mathsf{hashesSec}(V_{i-1}) \wedge \\ & \qquad \mathsf{cantReach}(V_{i-1},V_i) \wedge \mathsf{hashesIni}(V_{i-1},V_i)) \\ & \vee (\exists i \in [1,\ell]\ \mathsf{hashesCom}(V_i) \wedge \nexists \mathsf{id} : \texttt{*leaked-ind-secs}(\mathsf{id}, V_i)) \\ & \vee (\exists i \in [1,\ell]\ \mathsf{hashesJoiOnly}(V_i) \wedge \mathsf{conf}(V_i)) \\ & \vee (\exists i \in [1,\ell]\ \mathsf{guesses}(V_i)).\end{aligned}$$

The event that $\mathcal{A}$ can win.

We next look at the event that there exists a V with which $\mathcal{A}$ can win. Given the formula above, we have

$$\begin{aligned}\exists V \mathsf{hashesSec}(V) \wedge \mathsf{conf}(V) \implies & (\mathsf{hashesSec}(V_0) \wedge \mathsf{conf}(V_0)) & a) \\ & \vee (\exists V, U = \mathsf{par}(V) : \neg\mathsf{hashesSec}(U) \wedge \\ & \qquad \mathsf{cantReach}(U,V) \wedge \mathsf{hashesIni}(U,V)) & b) \\ & \vee (\exists V \mathsf{hashesCom}(V) \wedge \nexists \mathsf{id} : \texttt{*leaked-ind-secs}(\mathsf{id}, V)) & c) \\ & \vee (\exists V \mathsf{hashesJoiOnly}(V) \wedge \mathsf{conf}(V)) & d) \\ & \vee (\exists V \mathsf{guesses}(V)). & e)\end{aligned}$$

This means that $\mathcal{A}$'s advantage is upper-bounded by the sum of the probabilities of events *a)-e)*.

Lemma 4 (Bounding *a)*). *For any environment $\mathcal{A}$,*

$$\Pr[\mathsf{hashesSec}(V_0) \wedge \mathsf{conf}(V_0)] \le 1/2^\kappa.$$

Proof. We will show that if $\mathsf{conf}(V_0)$ is true, then the only secret which $\mathcal{A}$ can send to the RO to make $\mathsf{hashesSec}(V)$ true, i.e., the initParSec in V_0, is random and independent of $\mathcal{A}$'s view. Therefore, the probability of guessing it is at most $1/2^\kappa$, and this will conclude the proof.

Recall that, as per the `Initialization` operation of FREEK, the first epoch does not have epSec or joinerSec, and that the initParSec is chosen uniformly at random by the group creator. Moreover, the initParSec exists only in the state of the group creator and the only part of $\mathcal{A}$'s view that depends on it are the joiner secrets of children of V_0 (note that the joiner secrets are sent to the first members invited by the creator, and nothing used to derive the joiners leaves the creator's state). As these joiner secrets are outputs of the RO, they are independent of the corresponding inputs. Therefore, initParSec is independent of $\mathcal{A}$'s view unless $\mathcal{A}$ leaks it via a corruption of the group creator. However, this would contradict $\mathsf{conf}(V_0)$, and the proof is concluded. □

Lemma 5 (Bounding *c)*). *For any $\mathcal{A}$, there exists a reduction $\mathcal{B}$ s.t.*

$$\Pr[\exists V \,\mathsf{hashesCom}(V) \wedge \nexists \mathsf{id} : \texttt{*leaked-ind-secs}(\mathsf{id}, V)] \leq q_e \cdot \mathrm{Adv}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE}, q_e \log(q_n), q_n}(\mathcal{B}).$$

Proof. This is a straightforward consequence of Lemma H.7 from [AHKM22b]. □

Lemma 6 (Bounding *d)*). *For any $\mathcal{A}$, there exists a reduction $\mathcal{B}$ s.t.*

$$\Pr[\exists V \,\mathsf{hashesJoiOnly}(V) \wedge \mathsf{conf}(V)] \leq q_e \cdot \mathrm{Adv}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE}, 1, q_n}(\mathcal{B}).$$

Proof. This is a straightforward consequence of Lemma H.6 from [AHKM22b]. □

Lemma 7 (Bounding *e)*). *For any environment $\mathcal{A}$,*

$$\Pr[\exists V \,\mathsf{guesses}(V)] \leq \frac{s \cdot q_e \cdot q_h}{2^\kappa},$$

where s is the number of secrets in any epoch V.

Proof. Let $\mathcal{V} := \{V_1, \ldots, V_{q_e}\}$ be all epochs. Then,

$$\Pr[\exists V \,\mathsf{guesses}(V)] = \Pr\left[\bigvee_{i \in [q_e]} \mathsf{guesses}(V_i)\right] \leq \frac{s \cdot q_e \cdot q_h}{2^\kappa}.$$

□

Lemma 8 (Bounding *b)*). *For any $\mathcal{A}$, there exists a reduction $\mathcal{B}$ s.t.*

$$\Pr[\exists V, U = \mathsf{par}(V) : \neg\mathsf{hashesSec}(U) \wedge \mathsf{cantReach}(U, V) \wedge \mathsf{hashesIni}(U, V)] \\ \leq \mathrm{Adv}^{\mathsf{OW\text{-}PPRF}}_{\mathsf{PPRF}, q_e, q_m}(\mathcal{B}).$$

Here q_e is an upper bound on the total number of epochs and q_m is an upper bound on the number of children of a single epoch created in an execution with $\mathcal{A}$. The event E equal to $\exists V, U = \mathsf{par}(V) : \neg\mathsf{hashesSec}(U) \wedge \mathsf{cantReach}(U, V) \wedge \mathsf{hashesIni}(U, V)$ occurs when the predicate becomes true (the first time) in an execution with $\mathcal{A}$.

Proof. Let $\mathcal{A}$ be any environment. $\mathcal{B}$ simulates the experiment for $\mathcal{A}$ as defined in $\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}}, \mathcal{S}^2}$, lazily programming the RO. At a high level, the only difference from $\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}}, \mathcal{S}^2}$ concerns the key schedule derivation. That is, $\mathcal{B}$, embeds one PPRF key k from the $\mathsf{OW\text{-}PPRF}$ game it plays in the key schedule of each (non-injected) epoch. For operations involving k (which include programming the RO), $\mathcal{B}$ uses the Try, Corr and Eval oracles. Moreover, $\mathcal{B}$ embeds one challenge random value $r_{i,j}$ from the PPRF security game as the output of the RO on input $\mathsf{grpCtxt}_V()$ for one (honestly generated) epoch V. We will show that if the event E occurs, then $\mathcal{B}$ wins with one of its Try queries. Details follow.

Notation. Recall that in $\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}}, \mathcal{S}^2}$, each epoch U has an $\mathsf{initParSec}_U$, derived from its key schedule. Further, to derive the key schedule for a child V of U, the protocol computes the following:

$$\mathsf{joinerSec}_V = \mathsf{HKDF.Ext}(\mathsf{grpCtxt}_V(), \mathsf{initChildSec}_{U,V}, \mathsf{commitSec}_V), \text{ where}$$

$$\mathsf{initChildSec}_{U,V} = \mathsf{PPRF.eval}(\mathsf{initParSec}_U, \mathsf{HKDF.Exp}(\mathsf{grpCtxt}_V(), \mathsf{comSecConf}_V)).$$

Reduction $\mathcal{B}$

Key Schedule

When a new epoch V is created as a child of U, $\mathcal{B}$ executes FREEK, programming the RO if necessary, except the `*derive-keys` and `*derive-epoch-keys` methods of the key schedule is replaced by the following.

Case 1. V is honest, i.e., created on input Send.

1. Embed a PPRF instance in the key schedule of V — associate with V a new index i_V in the OW-PPRF game; k_{i_V} will be used instead of $\mathsf{initParSec}_V$.
2. If there was an index i_U associated with U, then embed a challenge in $\mathsf{initChildSec}_{U,V}$ — set the identifier V ($\mathsf{initChildSec}_{U,V}$ is the PPRF output on V) to some yet unused $r_{i_U,j}$ received from the game. Else, choose a random V.
3. Choose at random all hash outputs: $\mathsf{joinerSec}_V$, $\mathsf{comSecConf}_V$, $\mathsf{commitSec}_V$, appSec_V, $\mathsf{confTag}_V$ and $\mathsf{membKey}_V$.
4. Program the RO on correct inputs to output V and all outputs above except $\mathsf{joinerSec}_V$. The only RO links that re not programmed are: 1) The input to get $\mathsf{joinerSec}_V$ is unknown, as it contains the challenge PPRF output that allows $\mathcal{B}$ to win, 2) The output $\mathsf{initParSec}_V$ is unknown, as the instance is embedded there.

Case 2. V is injected, i.e., created when an id accepts an injected commit transitioning it to V.

1. Compute $\mathsf{comSecConf}_V$ and $\mathsf{commitSec}_V$ as in FREEK, programming the RO if necessary.
2. Search for a $\mathsf{joinerSec}_V$ that already appeared as output to some RO query that should be the joiner in V. That is, for each RO input equal to $(\mathsf{grpCtxt}_V(), y, \mathsf{commitSec}_V)$ for some y, run **CheckPPRF**(U, V, y). If a check succeeds (at most one can, since PPRF.eval is deterministic), let $\mathsf{joinerSec}_V$ denote the output. Else, $\mathsf{joinerSec}_V$ is not found.
3. If $\mathsf{joinerSec}_V$ is not found choose a random one.
4. Derive from $\mathsf{joinerSec}_V$ all secrets of V, including $\mathsf{initParSec}_V$ (by programming the RO).

Random Oracle

When $\mathcal{A}$ inputs an X for the first time, $\mathcal{B}$ programs the RO output Y computed as follows.

Case 1. $X = (\mathsf{epSec}_U, \textit{`init'})$, where epSec_U is the epoch secret of an epoch U with an associated i_U.

1. Query the oracle $\mathrm{Corr}(i_U)$, receive the unpunctured $\mathsf{initParSec}_U$, set $Y = \mathsf{initParSec}_U$.

Case 2. $X = (V, \mathsf{grpCtxt}_V(), y, \mathsf{commitSec}_V, \textit{`joiner'})$.

1. Check if a $\mathsf{joinerSec}_V$ matching the inputs has been already generated in **Key Schedule**. That is, check if some $\mathsf{joinerSec}_V$ was generated for V with parent U contained in $\mathsf{grpCtxt}_V()$, and **CheckPPRF**(U, V, y) is true.
2. If $\mathsf{joinerSec}_V$ has been generated, set Y to it. Else, choose Y at random.

Case 3. For any other input, pick Y at random.

Corruptions

The state of a party id contains for each epoch U on which id has a Move pebble, $\mathsf{initParSec}_U$ punctured on the $V_1, \ldots, V_\ell$ of all children V_i of U on which id does not have a Visited pebble. $\mathcal{B}$ computes the punctured init as follows.

1. If there is no index i_U associated with U, then the unpunctured $\mathsf{initParSec}_U$ is known to $\mathcal{B}$, so it punctures it on $V_1, \ldots, V_\ell$ itself.
2. Else, w.l.o.g. let $V_1 = r_{U,j_1}, \ldots, V_{\ell'} = r_{U,j_{\ell'}}$ denote those epoch id's that are equal to some challenges from the OW-PPRF game. Query $\mathrm{Corr}(i_U, j_1, \ldots, j_{\ell'})$ and receive $\mathsf{initParSec}'_U$.
3. Also puncture $\mathsf{initParSec}'_U$ on each $V_{\ell'+1}, \ldots, V_\ell$.

Procedure CheckPPRF(U, V, y)

// Returns true if $y = \mathsf{PPRF.eval}(\mathsf{initParSec}_U, V)$ for $\mathsf{initParSec}_U$ set to the parent init secret in U (possibly chosen by the OW-PPRF game).

1. If there is no index i_U associated with U, then $\mathsf{initParSec}_U$ is known to $\mathcal{B}$, so it simply evaluates the PPRF.
2. Else, let i_U be the index. If V is equal to some challenge $r_{i_U,j}$, query $\mathrm{Try}(i_U, j)$. Else, query $\mathrm{Eval}(i_U, V)$.

Figure 16: The reduction $\mathcal{B}$ for the proof of Lem. 8. Parts related to OW-PPRF oracles are marked in blue.

The reduction $\mathcal{B}$. There are three differences between the experiment emulated by $\mathcal{B}$ and $\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}$, each described in detail in Fig. 16. Roughly, the first difference is in the way the new key schedule is derived when an epoch is created in $\mathcal{F}_{\text{FR-CGKA}}$, i.e., when a party sends a packet, or receives an injected packet (by claim 1, the key schedule is the same for all parties, so $\mathcal{B}$ derives it only once). The second difference is in how $\mathcal{B}$ programs the RO. The third difference is in dealing with corruptions of parties holding init secrets.

The reduction $\mathcal{B}$ wins. It is easy to see that the experiment emulated by $\mathcal{B}$ is exactly the same as $\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}$ (note that the OW-PPRF game never aborts or refuses to reveal a value). Therefore, it remains to show that if the event E occurs, then $\mathcal{B}$ wins the OW-PPRF game. Recall that E occurs when there is an epoch V with parent U such that

1. $\mathcal{A}$ inputs to the RO $\mathsf{initChildSec}_{U,V}$,
2. $\mathcal{A}$ does not input to the RO any secret of U,
3. $\mathsf{cantReach}(U, V)$ is true, i.e., there is no id for which $\mathcal{A}$ can put a Move pebble on U without a Visited pebble on V.

It is easy to see that 3. implies that U was created via an honest Send, so there is an index i_U associated with it. Moreover, V was also created honestly, so its identifier V is a challenge $r_{i_U,j}$. Further, 1. implies that $\mathcal{A}$ inputs $y = \mathsf{PPRF.eval}(k_{i_U}, r_{i_U,j})$ to the RO. At this point $\mathcal{B}$ uses the Try oracle as part of the **CheckPPRF** procedure with the correct PPRF output y. It is left to show that 2. and 3. imply that this sets the win flag to true, i.e., that (i_U, j) is not in the corrupted set C in the OW-PPRF game.

Assume towards a contradiction that $(i_U, j) \in \mathsf{C}$. This means that $\mathcal{B}$ queried the Corr oracle on $(i_U, j_1, \ldots, j_\ell)$ where $j \notin \{j_1, \ldots, j_\ell\}$. Observe that $\mathcal{B}$ queries $\mathrm{Corr}(i_U, j_1, \ldots, j_\ell)$ in two cases.

a) When $\mathcal{A}$ sends epSec_U to the RO. This is a contradiction with 2.

b) When $\mathcal{A}$ corrupts an id with a Move pebble on U. In this case, $\mathcal{B}$ sets $\{j_1, \ldots, j_\ell\}$ to the indices of id's of those children of U on which id has a Visited pebble. Since j is not in the above set, id does not have a Visited pebble on V, which contradicts 3.

This concludes the proof of Lem. 8.

$\square$

Given Lems. 4 to 8, the proof of Lem. 3 is concluded. $\square$

$\square$

7.4 Authenticity

The fourth and final hybrid introduces authenticity, which is formalized by restoring the **auth** predicate. This hybrid matches the ideal experiment with $\mathcal{F}_{\text{FR-CGKA}}$. More concretely,

$\mathbf{Hyb}^4_{\mathcal{F}^4_{\text{FR-CGKA}},\mathcal{S}^4}$: The functionality $\mathcal{F}^4_{\text{FR-CGKA}}$ uses the original **auth** predicate from $\mathcal{F}_{\text{FR-CGKA}}$. The simulator $\mathcal{S}^4$ is the same as $\mathcal{S}^4$. The hybrid matches $\text{IDEAL}_{\mathcal{F}^4_{\text{FR-CGKA}},\mathcal{S}^4}$.

In the remainder of this section we show that, if Sig and MAC are unforgeable and if mmPKE is mmOW-RCCA secure, then FREEK guarantees authenticity, that is, the current hybrid is indistinguishable from the previous one. We note that security of mmPKE is needed e.g. to guarantee secrecy of MAC keys.

Proof intuition. We observe that the two hybrids are identical unless a bad event Forges occurs. Roughly, Forges happens if $\mathcal{A}$ breaks authenticity, that is, if it successfully impersonates an id_s towards id_r in an epoch epid such that **auth** is true for id_s in epid. Therefore, $\mathcal{A}$'s advantage in distinguishing the hybrids is upper bounded by the probability of Forges. This means that distinguishing between the two hybrids can be seen as a typical authenticity game, where the adversary wins by forging messages accepted by the protocol, as expressed by Forges.

Bad events. Let $\mathcal{A}$ be any environment. *Since epochs in detached trees are not authentic and the root cannot be injected, in the remainder of the proof we only consider non-root epochs in the main history-graph tree.*

The hybrids are identical unless Forges occurs. If it does, then the last assertion of `*assert-agree-auth-preserve` fails, implying the following event:

$$\exists V : \mathsf{Eps}[V].\mathsf{packet} = \text{`}inj\text{'} \wedge \mathbf{auth}(V, \mathsf{Eps}, \mathsf{Move}^{\mathcal{A}}, \mathsf{Key}^{\mathcal{A}}, \mathsf{Visited}^{\mathcal{A}}) \qquad (2)$$

For simplicity, in what follows we will omit the history graph Eps, and the pebbling sets $\mathsf{Move}^{\mathcal{A}}$, $\mathsf{Key}^{\mathcal{A}}$, $\mathsf{Visited}^{\mathcal{A}}$, from the authenticity predicate, and we will simply write $\mathbf{auth}(V)$.

As with the confidentiality predicate, **conf**, in the confidentiality proof, we observe that, given rules b) and c) of the pebbling-step validity, $\mathbf{auth}(V)$ implies one of the following. Below, U is the parent of V.

// $\mathcal{A}$ cannot reach a configuration from which it could put a move pebble on U for some id and id did not visit V

1. $\mathsf{cantReach}(U, V) \iff \nexists \mathsf{Config}^{\mathcal{A}}_{n'} : \mathsf{Config}^{\mathcal{A}} \vdash^* \mathsf{Config}^{\mathcal{A}}_{n'}$
$\wedge\ \exists \mathsf{id}(\mathsf{id}, U) \in \mathsf{Config}^{\mathcal{A}}_{n'}.\mathsf{Move}^{\mathcal{A}} \wedge (\mathsf{id}, V) \notin \mathsf{Config}^{\mathcal{A}}_{n'}.\mathsf{Visited}^{\mathcal{A}}$

// $\mathcal{A}$ cannot make a pebbling step

2. $\nexists \mathsf{id} : \texttt{*leaked-ind-secs}(\mathsf{id}, V)$.

By 2 and 1., 2., we derive the following:

$$\begin{aligned} \mathsf{Forges} \implies\ & \exists U, V (\mathsf{cantReach}(U) \wedge \mathsf{Eps}[V].\mathsf{packet} = \text{`}inj\text{'}) && a) \\ & \vee (\nexists \mathsf{id} : \texttt{*leaked-ind-secs}(\mathsf{id}, V) \wedge \mathsf{Eps}[V].\mathsf{packet} = \text{`}inj\text{'}) && b) \end{aligned}$$

where U is the parent of V. It thus remains to bound the probabilities of the events a) and b). By the safety predicate definition observe that b) takes place whenever there is a successful injection and the private signature keys of all ids in the target epoch are secure. Therefore b) reduces to the unforgeability of signatures. The idea is formalized in the following lemma.

Lemma 9. *For any adversary $\mathcal{A}$, there exists a reduction $\mathcal{B}$ s.t.*

$$\Pr[\exists U (\nexists \mathsf{id}\ \texttt{*leaked-ind-secs}(\mathsf{id}, V) \wedge \mathsf{Eps}[V].\mathsf{packet} = \text{`}inj\text{'})] \leq 2q_e \cdot \mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig}}(\mathcal{B}).$$

Proof. This is a straightforward consequence of Lemma H.10 from [AHKM22b]. □

Event a) is handled by the following lemma. We remind that, after a successful injection both the real-world and ideal-world executions stop, and we bound the probabilities of bad events under this assumption.

Lemma 10. *For any adversary $\mathcal{A}$, there exist reduction $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$ and $\mathcal{B}_4$ s.t.*

$$\begin{aligned} \Pr[\exists (U, V)\ \mathsf{cantReach}(U, V) \wedge \mathsf{Eps}[V].\mathsf{packet} = \text{`}inj\text{'}] \leq\ & \\ 1/2^{\kappa} + q_e \cdot \mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{B}_4) & \\ \mathrm{Adv}^{\mathsf{OW\text{-}PPRF}}_{\mathsf{PPRF}, q_e, q_m}(\mathcal{B}_1) + & \\ q_e \cdot \mathrm{Adv}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE}, q_e \log(q_n), q_n}(\mathcal{B}_2) + & \\ q_e \cdot \mathrm{Adv}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE}, 1, q_n}(\mathcal{B}_3) + \frac{s \cdot q_e \cdot q_h}{2^{\kappa}} & \end{aligned}$$

Proof. The event $\mathsf{cantReach}(U, V) \wedge \mathsf{Eps}[V].\mathsf{packet} = \text{‘}inj\text{’}$ occurs when the adversary manages for the first time to inject a message to a receiver id who is in epoch U, and either makes id transition to a new injected epoch V, or removes id from the group w.r.t. the injected epoch V. Note that, the injected epoch can also be created by $\mathcal{A}$ via a welcome message for V to a new group member, and then make other group members transition to that epoch.

Observe that $\mathsf{cantReach}(U, V)$ implies that, either the adversary doesn't have a move pebble on U, or if it does, then for all id such that $(\mathsf{id}, U) \in \mathsf{Move}^{\mathcal{A}}$, id has already visited V, i.e., it should be that $(\mathsf{id}, U) \in \mathsf{Visited}^{\mathcal{A}}$. Since V is a new, injected epoch, the latter cannot hold, thus $\mathsf{cantReach}(U, V)$ implies that $\mathcal{A}$ does not have a move pebble in U. In both cases $\mathbf{conf}(U)$ holds. More formally, we derive the following implication

$$\begin{aligned}\exists(U, V)\ \mathsf{cantReach}(U, V) \wedge \mathsf{Eps}[V].\mathsf{packet} = \text{‘}inj\text{’} \implies& \\ \exists(U, V)\ \mathbf{conf}(U) \wedge \mathsf{Eps}[V].\mathsf{packet} = \text{‘}inj\text{’}&\end{aligned}$$

therefore it suffices to bound the probability of the event on the right hand side of the above formula.

Similar to the confidentiality proof we define $\mathsf{hashesSec}(U)$ to be the following event: an input of $\mathcal{A}$ to RO contains epSec_V, $\mathsf{joinerSec}_V$ or $\mathsf{initParSec}_V$. Given the above, we compute

$$\begin{aligned}\Pr[\exists(U, V)\ \mathbf{conf}(U) \wedge \mathsf{Eps}[V].\mathsf{packet} = \text{‘}inj\text{’}] =& \\ \Pr[\exists(U, V)\ \mathbf{conf}(U) \wedge \mathsf{Eps}[V].\mathsf{packet} = \text{‘}inj\text{’} \wedge \mathsf{hashesSec}(U)]& \\ + \Pr[\exists(U, V)\ \mathbf{conf}(U) \wedge \mathsf{Eps}[V].\mathsf{packet} = \text{‘}inj\text{’} \wedge \neg\mathsf{hashesSec}(U)]&\end{aligned} \tag{3}$$

Observe that the following implications holds:

$$\begin{aligned}\exists(U, V)\ \mathbf{conf}(U) \wedge \mathsf{Eps}[V].\mathsf{packet} = \text{‘}inj\text{’} \wedge \mathsf{hashesSec}(U) \implies& \\ \exists U \mathbf{conf}(U) \wedge \mathsf{hashesSec}(U)&\end{aligned}$$

Given the above implication and the fact that the probability of the event on the right hand side has been already bounded by Lem. 3 in the confidentiality proof, we derive that there exist $\mathcal{B}_1$, $\mathcal{B}_2$, $\mathcal{B}_3$, such that

$$\begin{aligned}\Pr[\exists(U, V)\ \mathbf{conf}(U) \wedge \mathsf{Eps}[V].\mathsf{packet} = \text{‘}inj\text{’} \wedge \mathsf{hashesSec}(U)] \leq& \\ 1/2^{\kappa} + \mathrm{Adv}^{\mathsf{OW\text{-}PPRF}}_{\mathsf{PPRF},q_e,q_m}(\mathcal{B}_1) +& \\ q_e \cdot \mathrm{Adv}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE},q_e \log(q_n),q_n}(\mathcal{B}_2) +& \\ q_e \cdot \mathrm{Adv}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE},1,q_n}(\mathcal{B}_3) + \frac{s \cdot q_e \cdot q_h}{2^{\kappa}}&\end{aligned} \tag{4}$$

We finally show that there exists $\mathcal{B}_4$ such that

$$\begin{aligned}\Pr[\exists(U, V)\ \mathbf{conf}(U) \wedge \mathsf{Eps}[V].\mathsf{packet} = \text{‘}inj\text{’} \wedge \neg\mathsf{hashesSec}(U)] \leq& \\ q_e \cdot \mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{B}_4)&\end{aligned} \tag{5}$$

Since the adversary is not querying epSec_U to the RO and U is confidential, i.e., $\mathbf{conf}(U)$ holds, the MAC key, $\mathsf{membKey}$, which is the hash of epSec_U, is uniformly random and independent of $\mathcal{A}$'s view. Therefore, we prove the above bound via a straightforward reduction to the security of the MAC scheme. $\mathcal{B}_4$ acts as follows. First guesses the epoch that the adversary will try to inject to, i.e., it pics a uniformly random element i from the set $[q_e]$, and then simulates the execution of the hybrid, and only diverges as follows: for the i-th epoch it doesn't use $\mathsf{membKey}$ to compute MAC tags and forwards the input to the challenger and receives the tag, i.e., instead of computing

$\mathsf{tag}_t \leftarrow \mathsf{MAC.tag}(\gamma.\mathsf{membKey}, \gamma'.\mathsf{eid})$, it forwards $\gamma'.\mathsf{eid}$ to the challenger of the MAC unforgeability game, receives back tag_t and uses it in subsequent operations. MAC verification for messages of the i-th epoch are also forwarded to the challenger, while for the first pair $(\mathsf{tag}_t, \gamma'.\mathsf{eid})$ that verifies, but has not been generated by the challenger, $\mathcal{B}_4$ outputs that pair as a forgery to the challenger. Clearly, if this is the case, then U is a new injected epoch, $\mathcal{B}_4$ mounts a successful injection and the execution halts, assuming $\mathcal{B}_4$ correctly guesses the injected epoch, which happens with probability $1/q_e$. The bound of 5 follows.

By plugging, 5, 4, to 3, the proof of Lem. 10 is concluded, and this concludes FREEK's security proof.

□

8 Security Proof of O-FREEK

The following theorem formalizes the security of the O-FREEK protocol defined in Sec. 5.2.

Let $\mathcal{F}_{\text{FR-CGKA}}$ be the functionality from Fig. 8 with predicates **conf**, **auth**, defined in Fig. 9 with $P(\mathsf{Eps}, \mathsf{id}, V, \mathsf{Move}_0^{\mathcal{A}}, \mathsf{Key}_0^{\mathcal{A}}, \mathsf{Visited}_0^{\mathcal{A}})$ set to `false`. Let BtPke, BtSig, CR-PRF and Hash be the schemes instantiating O-FREEK. Denote the output of an environment $\mathcal{A}$ in the real execution with O-FREEK and the hybrid functionality $\mathcal{F}_{\text{AKS}}$ from Fig. 17 as $\text{REAL}_{\mathsf{O\text{-}FREEK}, \mathcal{F}_{\text{AKS}}}(\mathcal{A})$ and the output of $\mathcal{A}$ in the ideal execution with $\mathcal{F}_{\text{FR-CGKA}}$ and a simulator $\mathcal{S}$ as $\text{IDEAL}_{\mathcal{F}_{\text{FR-CGKA}}, \mathcal{S}}(\mathcal{A})$. Then, for any $\mathcal{A}$, there exists an $\mathcal{S}$ and adversaries $\mathcal{B}_1$ to $\mathcal{B}_5$ s.t.

$$\begin{aligned}\Pr[\text{IDEAL}_{\mathcal{F}_{\text{FR-CGKA}}, \mathcal{S}}(\mathcal{A}) = 1] - \Pr[\text{REAL}_{\mathsf{O\text{-}FREEK}, \mathcal{F}_{\text{AKS}}}(\mathcal{A}) = 1] \leq \\ \mathrm{Adv}^{\mathsf{CR}}_{\mathsf{Hash}}(\mathcal{B}_1) + \mathrm{Adv}^{\mathsf{CR}}_{\mathsf{CR\text{-}PRF}}(\mathcal{B}_2) \\ + 8q_e^2 q_n \cdot \mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{BtPke}}(\mathcal{B}_3) + 2q_e \cdot \mathrm{Adv}^{\mathsf{PRF}}_{\mathsf{CR\text{-}PRF}}(\mathcal{B}_4) \\ + 2q_e \cdot \mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{BtSig}}(\mathcal{B}_5),\end{aligned}$$

where q_e and q_n denote bounds on the number of epochs and the group size, respectively.

Proof. We prove our theorem using a sequence of hybrid experiments $\mathbf{Hyb}^i$ (transitioning from the real to the ideal execution), and we will use subscripts as with IDEAL to parameterize our hybrids with functionalities and simulators. For parts of the proof that are similar to those of FREEK, we will refer the reader to the corresponding parts of FREEK's proof.

$\mathbf{Hyb}^0$: This hybrid is equal to the real world execution, $\text{REAL}_{\mathsf{O\text{-}FREEK}, \mathcal{F}_{\text{AKS}}}(\mathcal{A})$.

$\mathbf{Hyb}^1_{\mathcal{F}^1_{\text{FR-CGKA}}, \mathcal{S}^1}$: The current hybrid is identical to $\mathbf{Hyb}^0$, but translated to the ideal-world language. The hybrid's definition and simulator $\mathcal{S}^1$ are as in $\mathbf{Hyb}^0$ of FREEK's proof, but using O-FREEK in the place of FREEK.

Lemma 11. *The view of $\mathcal{A}$ in $\mathbf{Hyb}^0$ is identical to the one in $\mathbf{Hyb}^1_{\mathcal{F}^1_{\text{FR-CGKA}}, \mathcal{S}^1}$.*

Proof. Straightforward by inspection. □ □

In the subsequent hybrids, we modify $\mathcal{F}^1_{\text{FR-CGKA}}$, which gives no security guarantees, by gradually introducing the security guarantees of FR-CGKA. We do so by gradually activating the security predicates and the checks (made by our helper functions via assertions), that enforce the properties of our protocol, i.e., consistency, step correctness, confidentiality and authenticity. In particular, $\mathcal{F}^2_{\text{FR-CGKA}}$ introduces consistency and step correctness, $\mathcal{F}^3_{\text{FR-CGKA}}$ introduces confidentiality, and $\mathcal{F}^4_{\text{FR-CGKA}}$ authenticity. The simulator is modified accordingly, to deal with less power given by the functionality. We first deal with consistency and step correctness.

8.1 Consistency and Step Correctness

Similar to FREEK, we show that O-FREEK guarantees step correctness and consistency, by showing indistinguishability between $\mathbf{Hyb}^1_{\mathcal{F}^1_{\text{FR-CGKA}},\mathcal{S}^1}$ and the second hybrid that we define below w.r.t. a new simulator $\mathcal{S}^2$ (the hybrid and simulator are as in FREEK; we recall them for the ease of exposition).

$\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}$: The current hybrid contains the functionality $\mathcal{F}^2_{\text{FR-CGKA}}$, which is the same as $\mathcal{F}_{\text{FR-CGKA}}$ except that it uses $\mathbf{conf} = \mathbf{auth} = \mathtt{false}$. The simulator $\mathcal{S}^2$ is defined as follows:

Simulator $\mathcal{S}^2$

$\mathcal{S}^2$ stores an O-FREEK state for each party id. When $\mathcal{F}^2_{\text{FR-CGKA}}$ sends to $\mathcal{S}^2$ an FR-CGKA input from id (for `Send`, `Receive`, `Join`, `GetKey`, or `DeleteMovePebble`), $\mathcal{S}^2$ runs O-FREEK on this input and updates the state. It sends to $\mathcal{F}^2_{\text{FR-CGKA}}$ the epoch identifier(s) U, V and well as any packets outputted by the protocol. For injected epochs, it sends to $\mathcal{F}^2_{\text{FR-CGKA}}$ the values sndr, act and mem taken from id's state. When id gets corrupted, $\mathcal{S}^2$ sends its state to the environment $\mathcal{A}$.

Indistinguishability between the current hybrid and the previous one is proven via the following lemma.

Lemma 12. *For any environment $\mathcal{A}$, there exists an adversary $\mathcal{B}$ such that*

$$\Pr\left[\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}(\mathcal{A}) \Rightarrow 1\right] - \Pr\left[\mathbf{Hyb}^1_{\mathcal{F}^1_{\text{FR-CGKA}},\mathcal{S}^1}(\mathcal{A}) \Rightarrow 1\right] \\ \leq \mathrm{Adv}^{\mathsf{CR}}_{\mathsf{Hash}}(\mathcal{B}) + \mathrm{Adv}^{\mathsf{CR}}_{\mathsf{CR\text{-}PRF}}(\mathcal{B}).$$

The probability runs over the randomness used by the experiments.

Proof. The main differences between the current hybrid ($\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}$) and the previous one ($\mathbf{Hyb}^1_{\mathcal{F}^1_{\text{FR-CGKA}},\mathcal{S}^1}$), are the following: (1) the outputs of the protocol are now set by the functionality $\mathcal{F}^2_{\text{FR-CGKA}}$ (instead of being directly set by the simulator as in $\mathbf{Hyb}^1_{\mathcal{F}^1_{\text{FR-CGKA}},\mathcal{S}^1}$), and (2) the execution can be disrupted if the checks made by $\mathcal{F}^2_{\text{FR-CGKA}}$ via **assert** statements and the helpers, fail. First we deal with (2).

Helper `*assert-agree-auth-preserved`. We show that the calls made to the helper `*assert-agree-auth-prese`
do not disrupt the execution. By the protocol definition, the epoch ids of newly generated epochs are computed as in FREEK, i.e., $V = \mathsf{Hash}(\gamma.\mathsf{grpCtxt}(), \mathsf{comSecConf})$, therefore the arguments made in FREEK's proof also hold for the current proof (assuming Hash is collision resistant), and calls to `*assert-agree-auth-preserved` do not disrupt the execution.

Helper `*step-correct`. We show that in $\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}$, if `*step-correct` is true when an id receives a message, then the protocol executed by $\mathcal{S}^2$ accepts the input. That is, the input refers to a packet that enables the transition from U to V, correctly extracted by $\mathcal{F}^2_{\text{FR-CGKA}}$. Notice that id's protocol rejects such a packet only if A) id does not have the epoch U in its state, B) it possesses the state information for the epoch U, but the secret key of the public-key encryption scheme has already been punctured on $\vec{V}$, or C) the correctness property of the signature or the encryption scheme fails, i.e., for an honestly generated packet, signature verification or decryption, fails. Regarding A), it only happens if id has never transitioned to U, or it executed DeleteMovePebble w.r.t. U. This implies that $(\mathsf{id}, U) \notin \mathsf{Move}$, which contradicts `*step-correct`. Regarding B), if the secret key of the encryption scheme is punctured on $\vec{V}$, given the fact that there are no cycles (as shown above), this means that id has already visited V and there cannot be another epoch with the same identifier, which again contradicts `*step-correct`. Finally, C) violates the (perfect) correctness of the either the encryption, or the signature, scheme.

Indistinguishability of outputs. As in FREEK, outputs that are chosen by the simulator are identical in the two hybrids. These include the epoch identifiers U, V, the packet C, and also, sndr and act, output by `Receive`, sndr and mem, output by `Join`, and the key, key, output by `GetKey`, for the cases in which the "if" statements in the operations are not satisfied (which implies that the target epoch is generated on-the-fly by the simulator as the operation executes). If the "if" statements are satisfied (which implies that the target epoch has been already created by a previous operation), then we need to make sure that the properties of the epoch (namely, sndr, act, mem, key) at the time of creation, match the ones returned to the parties in subsequent calls, i.e., the parties' CGKA states are consistent. We prove consistency of those values using similar arguments as with the main claim 1 in the proof of Lem. 2, for FREEK (and we conclude the proof of Lem. 12).

For sndr, act, mem, the idea is identical to that of FREEK: if the epoch identifier matches, then since $V = \mathsf{Hash}(\gamma.\mathsf{grpCtxt}_V(), \mathsf{comSecConf}_V)$ and $\gamma.\mathsf{grpCtxt}_V$ contains the last action and sender, sndr and act are the same for all parties that reach V (for all operations) assuming collision resistance of Hash. $\gamma.\mathsf{grpCtxt}_V$ also includes the tree hash, thus the member set outputted by `Join` is consistent with the tree in the states of other parties transitioning to the epoch. By the fact that the last action is the same and recursion, this is consistent with the member set in the functionality.

It remains to prove that the key output by `GetKey` is consistent. By collision resistance, all parties that reach V agree on $\gamma.\mathsf{grpCtxt}_V()$ and $\mathsf{comSecConf}_V$, which implies they also agree on r_V, since $\mathsf{comSecConf}_V = \mathsf{CR\text{-}PRF}(r_V, \text{'}\mathit{conf}\text{'})$ (by collision resistance of CR-PRF), therefore they all compute the same group key which is equal to $\gamma.\mathsf{appSec}_V \leftarrow \mathsf{CR\text{-}PRF}(r_V, \text{'}\mathit{app}\text{'})$.

This concludes the proof of Lemma 12.

$\square$

8.2 Confidentiality

The next hybrid introduces confidentiality, which is formalized by restoring the original confidentiality predicate of $\mathcal{F}_{\text{FR-CGKA}}$.

$\mathbf{Hyb}^3_{\mathcal{F}^3_{\text{FR-CGKA}},\mathcal{S}^3}$: The functionality $\mathcal{F}^3_{\text{FR-CGKA}}$ uses the original **conf** predicate from $\mathcal{F}_{\text{FR-CGKA}}$. The simulator $\mathcal{S}^3$ is the same as $\mathcal{S}^2$.

The indistinguishability between the current hybrid and the previous one is formalized in the following lemma.

Lemma 13. *For any environment $\mathcal{A}$, there exist adversaries $\mathcal{B}_1$ and $\mathcal{B}_2$ such that*

$$\Pr\left[\mathbf{Hyb}^3_{\mathcal{F}^3_{\text{FR-CGKA}},\mathcal{S}^3}(\mathcal{A}) \Rightarrow 1\right] - \Pr\left[\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}(\mathcal{A}) \Rightarrow 1\right] \\ \leq 8q_e^2 q_n \cdot \mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{BtPke}}(\mathcal{B}_1) + 2q_e \cdot \mathrm{Adv}^{\mathsf{PRF}}_{\mathsf{CR\text{-}PRF}}(\mathcal{B}_2).$$

Proof. We define a sequence of q_e hybrids $\mathsf{Hyb}_0, \ldots, \mathsf{Hyb}_{q_e}$, transitioning from $\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}$ to $\mathbf{Hyb}^3_{\mathcal{F}^3_{\text{FR-CGKA}},\mathcal{S}^3}$. The hybrids switch the evaluations of **conf** executed by $\mathcal{F}^2_{\text{FR-CGKA}}$ on input `GetKey` from `false` to the predicate from Fig. 9, one by one. That is, in Hyb_0 all evaluations use **conf** $=$ `false`, like in $\mathbf{Hyb}^2_{\mathcal{F}^2_{\text{FR-CGKA}},\mathcal{S}^2}$. In Hyb_1, the evaluation for the first epoch, in the order they are created, uses the predicate from Fig. 9, and so on. Clearly, Hyb_{q_e} is $\mathbf{Hyb}^3_{\mathcal{F}^3_{\text{FR-CGKA}},\mathcal{S}^3}$. Therefore, it remains to show that for each i there exist $\mathcal{B}_1$ and $\mathcal{B}_2$ such that

$$\begin{aligned}\Pr\left[\mathsf{Hyb}_{i+1}(\mathcal{A}) \Rightarrow 1\right] - \Pr\left[\mathsf{Hyb}_{i}(\mathcal{A}) \Rightarrow 1\right]& \\ \leq 8q_e q_n \cdot \mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{BtPke}}(\mathcal{B}_1) + 2 \cdot \mathrm{Adv}^{\mathsf{PRF}}_{\mathsf{CR\text{-}PRF}}(\mathcal{B}_2).&\end{aligned}$$

Let U denote the i-th created epoch. The only difference between Hyb_i and Hyb_{i+1} is as follows: In Hyb_i, the group key in U is computed (by the simulator) as $\mathsf{CR\text{-}PRF}(r, \text{‘}app\text{’})$. In Hyb_{i+1}, if **conf** from Fig. 9 is true, the group key in U is a uniformly random value chosen by the functionality independently of r. Moreover, recall that the only part of $\mathcal{A}$'s view that depends on r is the packet C outputted by the functionality (and generated by the simulator) when U is created. In particular, C contains (1) q_n ciphertexts, each encrypting r to one group member in U, and (2) $\mathsf{comSecConf}$ equal to $\mathsf{CR\text{-}PRF}(r, \text{‘}conf\text{’})$.

We next define a sequence of $2q_n + 1$ hybrids, $\mathsf{Hyb}_{i,0}$ to $\mathsf{Hyb}_{i,2q_n+1}$ transitioning from Hyb_i to Hyb_{i+1}. Assume for a moment that **conf** from Fig. 9 is always true in U (we later show how to remove this assumption).

Switching ciphertexts. The first q_n hops replace, one by one, the encryptions of r in C by encryptions of a random and independent value s. That is, in $\mathsf{Hyb}_{i,0}$ all ciphertexts encrypt r like in Hyb_i. In $\mathsf{Hyb}_{i,1}$, the first ciphertext (i.e. for the lexicographically smallest public key) encrypts a random and independent s. In $\mathsf{Hyb}_{i,2}$, the first two ciphertexts encrypt the same random and independent s, ans so on.

Switching the group key. Observe that in Hyb_{i,q_n}, all ciphertexts in C encrypt s. This means that r used to derive the group key as $\mathsf{CR\text{-}PRF}(r, \text{‘}app\text{’})$ and $\mathsf{comSecConf} = \mathsf{CR\text{-}PRF}(r, \text{‘}conf\text{’})$ is independent of $\mathcal{A}$'s view, apart from the above derivations. Therefore, we can use the PRF property to switch $\mathsf{CR\text{-}PRF}(r, \text{‘}app\text{’})$ to a random and independent group key in hop $q_n + 1$.

Switching ciphertexts back. Finally, observe that the only difference between Hyb_{i,q_n+1} and Hyb_{i+1} is that in the former C encrypts a random and independent s, while in the latter it encrypts r which is also used to compute $\mathsf{comSecConf} = \mathsf{CR\text{-}PRF}(r, \text{‘}conf\text{’})$. Therefore, we define the last q_n hops which revert the first q_n hops — they switch the encryptions of s by encryptions of r, one by one.[14]

Removing the assumption that conf in U is true. In general, for each j, $\mathsf{Hyb}_{i,j}$ should switch anything (at the time C is generated) only if **conf** in U is true at the time `GetKey` in U is called. Else, it should behave as Hyb_i. The reason is that if something was switched and later a party id was corrupted making U not confidential, then hybrids would be trivially distinguishable. E.g., a secret key in id's state would allow to decrypt an s different than r used to compute $\mathsf{comSecConf}$.

To make this possible, we define hybrids Hyb'_i and Hyb'_{i+1} which are the same as Hyb_i and Hyb_{i+1}, resp., except in Hyb'_i and Hyb'_{i+1} the environment $\mathcal{A}'$ commits at the beginning to a bit indicating if U is confidential at the time `GetKey` in U is called. Technically, the experiment stops as soon as the bit announced by $\mathcal{A}'$ turns out to be incorrect. Observe that for any $\mathcal{A}$ with advantage ϵ in distinguishing Hyb_i and Hyb_{i+1}, there is a trivial $\mathcal{A}'$ with advantage $\epsilon/2$ in distinguishing Hyb'_i and

[14]One may hope to get a tighter reduction that does not need the last q_n hops by having the simulator $\mathcal{S}^3$ choose $\mathsf{comSecConf}$ at random instead. However, this doesn't work with adaptive corruptions – $\mathcal{S}^3$ has to commit to whether to make $\mathsf{comSecConf}$ random or not at the moment the epoch is created. If it chooses $\mathsf{comSecConf}$ at random and $\mathcal{A}$ later corrupts a party, $\mathcal{S}^3$ loses. If it chooses real and the epoch ends up confidential, our reduction doesn't work. Guessing the strategy for *all* epochs would make $\mathcal{S}^3$ inefficient.

Hyb'_{i+1} — $\mathcal{A}'$ guesses the bit, runs $\mathcal{A}$ and outputs what $\mathcal{A}$ outputs if the guessed bit is correct, and 0 otherwise.[15] Therefore, it remains to upper-bound the advantage of an adversary $\mathcal{A}'$ in distinguishing Hyb'_i and Hyb'_{i+1}.

To this end, we define hybrids $\mathsf{Hyb}'_{i,0}$ to $\mathsf{Hyb}'_{i,2q_n+1}$ transitioning from Hyb'_i to Hyb'_{i+1} as follows. For each j, $\mathsf{Hyb}'_{i,j}$ is the same as $\mathsf{Hyb}_{i,j}$ if $\mathcal{A}'$ announces that U will be confidential. Else, $\mathsf{Hyb}_{i,j}$ is the same as Hyb_i.

To conclude the proof, it is left to prove the following two claims.

Claim 2. *For any adversary $\mathcal{A}'$ and any $i \in [q_e], j \in [q_n] \cup [q_n+2, 2q_n+2]$, there exists an adversary $\mathcal{B}$ s.t.*

$$\Pr\left[\mathsf{Hyb}'_{i,j+1}(\mathcal{A}') \Rightarrow 1\right] - \Pr\left[\mathsf{Hyb}'_{i,j}(\mathcal{A}') \Rightarrow 1\right] \leq 2q_e \cdot \mathrm{Adv}^{\mathsf{IND\text{-}CCA}}_{\mathsf{BtPke}}(\mathcal{B}).$$

Claim 3. *For any adversary $\mathcal{A}$, and any $i \in [q_e]$ there exists an adversary $\mathcal{B}$ s.t.*

$$\Pr\left[\mathsf{Hyb}'_{i,q_n+1}(\mathcal{A}') \Rightarrow 1\right] - \Pr\left[\mathsf{Hyb}'_{i,q_n}(\mathcal{A}') \Rightarrow 1\right] \leq \mathrm{Adv}^{\mathsf{PRF}}_{\mathsf{CR\text{-}PRF}}(\mathcal{B}).$$

of the first claim. We only consider the first sequence $i \in [q_e], j \in [q_n]$, as the second one is analogous. Notice that Hyb'_{i,q_n} and $\mathsf{Hyb}'_{i+1,0}$ are the same. Therefore, it remains to consider $\mathsf{Hyb}'_{i,j}$ and $\mathsf{Hyb}'_{i,j+1}$ for all j.

Recall that the only difference between $\mathsf{Hyb}'_{i,j}$ and $\mathsf{Hyb}'_{i,j+1}$ is the message contained in the j-th ciphertext c_j in the packet creating the i-th epoch in the order of creation. In the first hybrid c_j encrypts the "real" value r, and in the latter – a random s.

Let $\mathcal{A}'$ be any adversary. The reduction $\mathcal{B}$ receives epk from the $\mathsf{IND\text{-}CCA}$ game and runs $\mathcal{A}'$. In general, $\mathcal{B}$ runs the code of the functionality and the simulator as in $\mathsf{Hyb}'_{i,j}$. The only difference is that if the i-th epoch will be safe, $\mathcal{B}$ makes the following changes.

1. *Embed* epk. Recall that $\mathsf{O\text{-}FREEK}$ generates key pairs (only) when it creates an epoch — on each such operation, it generates a key pair for the sender and, in case this is an add operation, for the added member (technically, the latter is done by $\mathcal{F}_{\text{AKS}}$ at this exact moment). Therefore, $\mathcal{B}$ guesses an $e \in [q_e]$ and a bit b and embeds epk when the e-th epoch is created, as one of the two keys according to b. Let id^* denote the party for which epk is generated.

2. *Embed challenge.* When the i-th epoch is created, $\mathcal{B}$ sends to the challenger r and s as well as the identity vector $I^* = \mathsf{BtIds}[\mathsf{id}^*]$ of id^* used by the sender (emulated by $\mathcal{B}$). It receives c^* and uses it instead of c_j.

3. For operations using esk, that is, receiving messages by id^* and computing its state when corrupted, $\mathcal{B}$ uses the Corr and Dec oracles. The only exception is when id^* receives c^* in the i-th epoch, $\mathcal{B}$ runs its protocol with r directly.

$\mathcal{B}$ outputs whatever $\mathcal{A}'$ outputs. Assume $\mathcal{B}$'s guess is correct. We next show that in this case the **req** checks executed by the $\mathsf{IND\text{-}CCA}$ game pass. Once this is proved, it is easy to see that $\mathcal{B}$'s emulation is $\mathsf{Hyb}'_{i,j}$ if the challenge bit is 0 and $\mathsf{Hyb}'_{i,j+1}$ otherwise. This will conclude the proof.

The game makes two **req** checks. The first one happens in the Dec oracle. $\mathcal{B}$ uses Dec as in Step 3 above. When id^* receives a message transitioning to the i-th epoch, $\mathcal{B}$ does not use Dec on c^*, so the **req**'ed condition is not violated. When id^* receives a message transitioning to another epoch V, the

[15]The advantage of $\mathcal{A}'$ is a trivial calculation after noticing that the guessed bit is independent of the view of $\mathcal{A}$.

identity $\vec{I}$ used for decryption will be different than $\vec{I}^*$ — indeed, $\vec{I}$ ends with the unique identifier of V.

The second check happens at the end of the game. For this, we show that if **conf** is true for the i-th epoch, then $\mathcal{B}$'s corruptions do not violate the **req**'ed condition. Assume towards a contradiction that $\mathcal{B}$ violates the condition. This means that $\mathcal{B}$ queries Corr on $\vec{I}$ which is a prefix of $\vec{I}^*$. We next show that this allows the adversary to put a key pebble on the i-th epoch which is a contradiction.

Recall that for each epoch U_m on which id^* has a move pebble, it stores a bunch of keys for identity vectors $(U_1, 0, \ldots, U_m, 0, x)$ where $U_1, \ldots, U_m$ identify ancestors of U_m and x identifies a node in the binary tree $\mathsf{eskTree}$ on which the key has not been punctured. If id^* has a visited pebble on a child U_{m+1} of U_m, then its state contains no keys for identity vectors where x is a prefix of U_{m+1}.

Now let $U_1, \ldots, U_n$ denote the ancestors of the i-th epoch, U_n, ordered from the oldest. By the above observation, if $\vec{I}$ is a prefix of $\vec{I}^*$, then at the moment of corruption id^* has a move pebble on a U_m for $m < n$ and it does not have a visited pebble on U_{m+1}. Therefore, the adversary also gets these pebbles for id. Moreover, no epoch $U_1, \ldots, U_n$ removes or is created by id^* as this would clear the identity vector. Therefore, the adversary can put move and key pebbles on U_{m+1}. This, in turn, allows it to put a pebble on U_{m+2}, and so on, resulting in a key pebble on the i-th epoch. This contradicts **conf** and concludes the proof.

□

of the second claim. In both hybrids, if the i-th epoch will be confidential, the key for $\mathsf{CR\text{-}PRF}$ in epoch i is random and independent of the view of $\mathcal{A}'$. Therefore, we construct a straightforward reduction that runs $\mathcal{A}'$, emulating the code of the simulator and the functionality, except if the i-th epoch is secure it does as follows. First, it queries the Eval oracle on *'conf'* to get the $\mathsf{comSecConf}$ for the i-th epoch. Second, it gets the challenge on input *'app'* and uses it as the group key. □

□ □

8.3 Authenticity

The fourth and final hybrid introduces authenticity, which is formalized by restoring the **auth** predicate. This hybrid matches the ideal experiment with $\mathcal{F}_{\text{FR-CGKA}}$. More concretely,

$\mathbf{Hyb}^4_{\mathcal{F}^4_{\text{FR-CGKA}}, \mathcal{S}^4}$: The functionality $\mathcal{F}^4_{\text{FR-CGKA}}$ uses the original **auth** predicate from $\mathcal{F}_{\text{FR-CGKA}}$. The simulator $\mathcal{S}^4$ is the same as $\mathcal{S}^4$. The hybrid matches $\text{IDEAL}_{\mathcal{F}^4_{\text{FR-CGKA}}, \mathcal{S}^4}$.

In the remainder of this section we show that, if Sig is unforgeable, then $\mathsf{O\text{-}FREEK}$ guarantees authenticity, that is, the current hybrid is indistinguishable from the previous one. More formally,

Lemma 14. *For any environment $\mathcal{A}$, there exist an adversary $\mathcal{B}$ such that*

$$\Pr\left[\mathbf{Hyb}^4_{\mathcal{F}^4_{\text{FR-CGKA}}, \mathcal{S}^4}(\mathcal{A}) \Rightarrow 1\right] - \Pr\left[\mathbf{Hyb}^3_{\mathcal{F}^3_{\text{FR-CGKA}}, \mathcal{S}^3}(\mathcal{A}) \Rightarrow 1\right] \\ \leq 2q_e \cdot \mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{BtSig}}(\mathcal{B}).$$

Proof. Let $\mathcal{A}$ be any environment. Observe that the hybrids are identical unless the following bad event Forges happens:

$$\exists V : \mathsf{Eps}[V].\mathsf{packet} = \text{'}\mathit{inj}\text{'} \wedge \mathbf{auth}(V, \mathsf{Eps}, \mathsf{Move}^{\mathcal{A}}, \mathsf{Key}^{\mathcal{A}}, \mathsf{Visited}^{\mathcal{A}})$$

Therefore, it remains to upper-bound the probability of Forges. To this end, next construct a reduction $\mathcal{B}$ such that $\Pr[\mathsf{Forges}] \leq 2q_e \cdot \mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{BtSig}}(\mathcal{B})$.

The reduction $\mathcal{B}$ receives spk from the $\mathsf{EUF\text{-}CMA}$ game and runs $\mathcal{A}$. In general, $\mathcal{B}$ runs the code of the functionality and the simulator as in $\mathbf{Hyb}^3_{\mathcal{F}^3_{\text{FR-CGKA}},\mathcal{S}^3}$, except the following.

1. *Embed* spk. Recall that $\mathsf{O\text{-}FREEK}$ generates key pairs (only) when it creates an epoch — on each such operation, it generates a key pair for the sender and, in case this is an add operation, for the added member (technically, the latter is done by $\mathcal{F}_{\text{AKS}}$ at this exact moment). Therefore, $\mathcal{B}$ guesses an $e \in [q_e]$ and a bit b and embeds spk when the e-th epoch is created, as one of the two keys according to b. Let id^* denote the party for which spk is generated.

2. For operations using spk, i.e., sending messages by id^* and computing its state when corrupted, $\mathcal{B}$ uses Sig, Upd and Corr oracles in the natural way.

3. *Get the forgery.* If Forges happens, $\mathcal{B}$ halts and sends to the challenger the following forgery: Let V^* be the epoch that makes Forges happen. Let id_r be the first party who transitioned there and let c^* be the (injected) packet it used. $\mathcal{B}$'s forgery consists of σ^* contained in c^*, the message V^* and the identity vector $\vec{I}^*$ used by id_r.

It remains to show that if Forges occurs and $\mathcal{B}$'s guess is correct, i.e., the injection happens on behalf of id^* when it uses spk, then $\mathcal{B}$ wins the game. Assume towards a contradiction that $\mathcal{B}$ loses. This means that one of the three condition s checked by the game to compute the win flag fails.

The first condition is false if $\mathsf{Sig.vrf}$ outputs 0. But id_r accepted c^*, which means that its protocol run $\mathsf{Sig.vrf}$ with result 1. Verification is deterministic, so we get a contradiction.

The second condition is false if $\mathcal{B}$ queried V^* to the Sig oracle. However, $\mathcal{B}$ only uses Sig with epochs *honestly* created by id^*. Since Forges implies that V^* is not honest (formally, its $\mathsf{packet} \neq$ *'inj'*), we get a contradiction.

The third and final condition is false if $\mathcal{B}$ queried Corr and received a secret key for $\vec{I_c} \neq \perp$ s.t. there is an $\vec{I'}$ s.t. $\vec{I}^* = \vec{I_c} \parallel \vec{I}^*$. We will how that this allows the adversary to put a move pebble on the parent of V^*, which is a contradiction with **auth**.

ecall that for each epoch U_m on which id^* has a move pebble, it stores a bunch of secret keys for identity vectors $(U_1, 0, \ldots, U_m, 0, x)$ where $U_1, \ldots, U_m$ identify ancestors of U_m and x identifies a node in the binary tree $\mathsf{sskTree}$ on which the key has not been punctured. If id^* has a visited pebble on a child U_{m+1} of U_m, then its state contains no keys for identities where x is a prefix of U_{m+1}.

Now let $U_1, \ldots, U_n$ denote the ancestors of V^*, ordered from the oldest to the parent U_n of V^*. By the above observation, if $\vec{I}$ is a prefix of $\vec{I}^*$, then at the moment of corruption id^* has a move pebble on a U_m. If $m = n$, then the adversary has a move pebble on the parent of V^* and we are done. Else, again by the observation, id^* does not have a visited pebble on U_{m+1}. Therefore, the adversary can put a move pebble on U_{m+1} and remove visited pebbles from its children. This in turn allows it to do the same for U_{m+2} and so on. Eventually, the adversary gets a move pebble on U_n, which concludes the proof. □

□

References

[AAB$^+$21] Joël Alwen, Benedikt Auerbach, Mirza Ahad Baig, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. Grafting key trees: Efficient key management for overlapping groups. In Kobbi Nissim and Brent Waters, editors, *TCC 2021, Part III*, volume 13044 of *LNCS*, pages 222–253. Springer, Heidelberg, November 2021.

[AAN$^+$22a] Joël Alwen, Benedikt Auerbach, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, and Krzysztof Pietrzak. DeCAF: Decentralizable continuous group key agreement with fast healing. Cryptology ePrint Archive, Report 2022/559, 2022. `https://eprint.iacr.org/2022/559`.

[AAN$^+$22b] 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, Heidelberg, May / June 2022.

[ACC$^+$21] Joël Alwen, Margarita Capretto, Miguel Cueto, Chethan Kamath, Karen Klein, Ilia Markov, Guillermo Pascual-Perez, Krzysztof Pietrzak, Michael Walter, and Michelle Yeo. Keep the dirt: Tainted treekem, an efficient and provably secure continuous group key agreement protocol. 42nd IEEE Symposium on Security and Privacy, 2021. Full Version: `https://ia.cr/2019/1489`.

[ACDJ23] M. R. Albrecht, S. Celi, B. Dowling, and D. Jones. Practically-exploitable cryptographic vulnerabilities in matrix. In *2023 2023 IEEE Symposium on Security and Privacy (SP) (SP)*, pages 1419–1436, Los Alamitos, CA, USA, may 2023. IEEE Computer Society.

[ACDT20] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. Security analysis and improvements for the IETF MLS standard for group messaging. In Daniele Micciancio and Thomas Ristenpart, editors, *CRYPTO 2020, Part I*, volume 12170 of *LNCS*, pages 248–277. Springer, Heidelberg, August 2020.

[ACDT21] 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.

[ACJM20] 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, Heidelberg, November 2020.

[AHKM22a] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. Server-aided continuous group key agreement. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 69–82. ACM Press, November 2022.

[AHKM22b] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. Server-aided continuous group key agreement. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security*, CCS '22, page 69–82, New York, NY, USA, 2022. Association for Computing Machinery.

[AJM22] 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, Heidelberg, August 2022.

[Aut23] Automerge.org. Automerge, 2023. `https://automerge.org/`.

[BB$^+$20] R. Barnes, B. Beurdouche, , J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. The messaging layer security (mls) protocol (draft-ietf-mls-protocol-latest). Technical report, IETF, Oct 2020. `https://messaginglayersecurity.rocks/mls-protocol/draft-ietf-mls-protocol.html`.

[BBR$^+$22] Richard Barnes, Benjamin Beurdouche, Raphael Robert, Jon Millican, Emad Omara, and Katriel Cohn-Gordon. The Messaging Layer Security (MLS) Protocol. Internet-Draft draft-ietf-mls-protocol-17, Internet Engineering Task Force, December 2022. Work in Progress.

[BCK21] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Cryptographic security of the MLS RFC, draft 11. Cryptology ePrint Archive, Report 2021/137, 2021. `https://eprint.iacr.org/2021/137`.

[BCK22] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Security analysis of the MLS key derivation. In *2022 IEEE Symposium on Security and Privacy*, pages 2535–2553. IEEE Computer Society Press, May 2022.

[BCV22] David Balbás, Daniel Collins, and Serge Vaudenay. Cryptographic administration for secure group messaging. Cryptology ePrint Archive, Report 2022/1411, 2022. `https://eprint.iacr.org/2022/1411`.

[BDG$^+$22] Alexander Bienstock, Yevgeniy Dodis, Sanjam Garg, Garrison Grogan, Mohammad Hajiabadi, and Paul Rösler. On the worst-case inefficiency of CGKA. In Eike Kiltz and Vinod Vaikuntanathan, editors, *TCC 2022, Part II*, volume 13748 of *LNCS*, pages 213–243. Springer, Heidelberg, November 2022.

[BDT22] Alexander Bienstock, Yevgeniy Dodis, and Yi Tang. Multicast key agreement, revisited. In Steven D. Galbraith, editor, *CT-RSA 2022*, volume 13161 of *LNCS*, pages 1–25. Springer, Heidelberg, March 2022.

[BGI14] Elette Boyle, Shafi Goldwasser, and Ioana Ivan. Functional signatures and pseudorandom functions. In Hugo Krawczyk, editor, *PKC 2014*, volume 8383 of *LNCS*, pages 501–519. Springer, Heidelberg, March 2014.

[BMO$^+$18] Richard Barnes, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. Message layer security (mls) wg. https://datatracker.ietf.org/wg/mls/about/, 2018.

[But14] Vitalik Buterin. Ethereum: A next-generation smart contract and decentralized application platform, 2014.

[BW13] Dan Boneh and Brent Waters. Constrained pseudorandom functions and their applications. In Kazue Sako and Palash Sarkar, editors, *ASIACRYPT 2013, Part II*, volume 8270 of *LNCS*, pages 280–300. Springer, Heidelberg, December 2013.

[CCG16] Katriel Cohn-Gordon, Cas J. F. Cremers, and Luke Garratt. On post-compromise security. In *IEEE 29th Computer Security Foundations Symposium, CSF 2016, Lisbon, Portugal, June 27 - July 1, 2016*, pages 164–178. IEEE Computer Society, 2016.

[CEST22] Kelong Cong, Karim Eldefrawy, Nigel P. Smart, and Ben Terner. The key lattice framework for concurrent group messaging. Cryptology ePrint Archive, Report 2022/1531, 2022. `https://eprint.iacr.org/2022/1531`.

[CPZ20] Melissa Chase, Trevor Perrin, and Greg Zaverucha. The signal private group system and anonymous credentials supporting efficient verifiable encryption. In Jay Ligatti, Xinming Ou, Jonathan Katz, and Giovanni Vigna, editors, *ACM CCS 2020*, pages 1445–1459. ACM Press, November 2020.

[DDF21] 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, Heidelberg, October 2021.

[Fou23a] The Matrix.org Foundation. Matrix specification, 2023. `https://spec.Matrix.org/v1.6`.

[Fou23b] The Matrix.org Foundation. Matrix state resolution, 2023. `https://spec.Matrix.org/v1.6/rooms/v10`.

[GGM84] 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.

[GM15] Matthew D. Green and Ian Miers. Forward secure asynchronous messaging from puncturable encryption. In *2015 IEEE Symposium on Security and Privacy*, pages 305–320. IEEE Computer Society Press, May 2015.

[Gmb21] Wire Swiss GmbH. Wire security whitepaper, 2021. `https://wire-docs.wire.com/download/Wire+Security+Whitepaper.pdf`.

[Goo23] Google. Google docs, 2023. `https://docs.google.com/`.

[GS02] Craig Gentry and Alice Silverberg. Hierarchical ID-based cryptography. In Yuliang Zheng, editor, *ASIACRYPT 2002*, volume 2501 of *LNCS*, pages 548–566. Springer, Heidelberg, December 2002.

[HKP$^+$21] 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.

[HKP22] Keitaro Hashimoto, Shuichi Katsumata, and Thomas Prest. How to hide MetaData in MLS-like secure group messaging: Simple, modular, and post-quantum. In Heng Yin, Angelos Stavrou, Cas Cremers, and Elaine Shi, editors, *ACM CCS 2022*, pages 1399–1412. ACM Press, November 2022.

[HLA19] Chris Howell, Tom Leavy, and Joël Alwen. Wickr messaging protocol : Technical paper, 2019. `https://1c9n2u3hx1x732fbvk1ype2x-wpengine.netdna-ssl.com/wp-content/uploads/2019/12/WhitePaper_WickrMessagingProtocol.pdf`.

[Jab23] Jabber. Jabber, 2023. `https://www.jabber.org/`.

[KEO$^+$22] Kaisei Kajita, Keita Emura, Kazuto Ogawa, Ryo Nojima, and Go Ohtake. Continuous group key agreement with flexible authorization and its applications. Cryptology ePrint Archive, Report 2022/1768, 2022. `https://eprint.iacr.org/2022/1768`.

[KKPP20] Shuichi Katsumata, Kris Kwiatkowski, Federico Pintore, and Thomas Prest. Scalable ciphertext compression techniques for post-quantum KEMs and their applications. In Shiho Moriai and Huaxiong Wang, editors, *ASIACRYPT 2020, Part I*, volume 12491 of *LNCS*, pages 289–320. Springer, Heidelberg, December 2020.

[KPTZ13] 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.

[Mat23a] Matrix.org. are we mls yet?, 2023. `http://arewemlsyet.com/`.

[Mat23b] Matrix.org. Decentralised mls, 2023. `https://gitlab.matrix.org/matrix-org/mls-ts/-/blob/decentralised2/decentralised.org`.

[MP16] M. Marlinspike and T. Perrin. The double ratchet algorithm, 11 2016. `https://whispersystems.org/docs/specifications/doubleratchet/doubleratchet.pdf`.

[MP22] M. Marlinspike and T. Perrin. Signal – technical information, 2022. `https://signal.org/docs/`.

[Nak08] Satoshi Nakamoto. Bitcoin: A peer-to-peer electronic cash system, Dec 2008.

[OR93] J. Oikarinen and D. Reed. Internet relay chat protocol. RFC 1459, RFC Editor, 1993.

[Tea23] Microsoft Teams. Group chat software, 2023. `https://www.microsoft.com/en-us/microsoft-teams/group-chat-software`.

[Wei19] Matthew Weidner. Group messaging for secure asynchronous collaboration. MPhil Dissertation, 2019. Advisors: A. Beresford and M. Kleppmann, 2019. `https://mattweidner.com/acs-dissertation.pdf`.

[Wha23] WhatsApp. Whatsapp encryption overview, 2023. `https://z-p3-scontent-dub4-1.xx.fbcdn.net/v/t39.8562-6/328495424_498532869106467_756303412205949548_n.pdf?_nc_cat=104&ccb=1-7&_nc_sid=ad8a9d&_nc_ohc=hXI5qoXRQZYAX_07jvi&_nc_ht=z-p3-scontent-dub4-1.xx&oh=00_AfA_WYI48K0JHrfvQ12DfuJYRtocrVeZ8JG_StJPWqKIzQ&oe=63F4133C`.

[WKHB21] Matthew Weidner, Martin Kleppmann, Daniel Hugenroth, and Alastair R. Beresford. Key agreement for decentralized secure group messaging with strong security guarantees. In Giovanni Vigna and Elaine Shi, editors, *ACM CCS 2021*, pages 2024–2045. ACM Press, November 2021.

[WPBB22] Théophile Wallez, Jonathan Protzenko, Benjamin Beurdouche, and Karthikeyan Bhargavan. TreeSync: Authenticated group management for messaging layer security. Cryptology ePrint Archive, Report 2022/1732, 2022. `https://eprint.iacr.org/2022/1732`.

A Security Model Intuition

For completeness, we include the intuition for the security model of [AHKM22b]. This text is copied verbatim from [AHKM22b].

Security of saCGKA protocols is defined in the UC framework. This means that we define a real-world experiment where an environment $\mathcal{A}$ interacts with a saCGKA protocol π and an ideal-world experiment where $\mathcal{A}$ interacts with an ideal saCGKA functionality $\mathcal{F}_{\text{FR-CGKA}}$ and a simulator $\mathcal{S}$. A protocol π is secure if for all $\mathcal{A}$ there exists an $\mathcal{S}$ such that the difference between the probability that $\mathcal{A}$ outputs 1 the real world with π and the probability that $\mathcal{A}$ outputs 1 in the ideal world with $\mathcal{F}_{\text{FR-CGKA}}$ and $\mathcal{S}$ are negligible.

Readers familiar with game-based security should think of $\mathcal{A}$ as the adversary attacking the protocol. The simulator $\mathcal{S}$ models all parts of the protocol irrelevant for security.

We next look at both worlds for the specific case of saCGKA.

The real world. In the real-world experiment, the following actions are available to $\mathcal{A}$: First, it can instruct parties to perform different group operations, creating new epochs. When this happens, the party runs the protocol, updates its state and hands to $\mathcal{A}$ the message meant to be sent to the mailboxing service. The mailboxing service is fully controlled by $\mathcal{A}$. This means that the next action it can perform is to deliver arbitrary messages to parties. A party receiving such message updates its state (or creates it in case of new members) and notifies $\mathcal{A}$ which group operation it applied. Moreover, $\mathcal{A}$ can fetch from parties group keys computed according to their current states and corrupt them by exposing their current states.[16]

The ideal world. In the ideal-world experiment $\mathcal{A}$ can perform the same actions, but instead of the protocol, parties use the ideal CGKA functionality, $\mathcal{F}_{\text{FR-CGKA}}$, and a simulator $\mathcal{S}$. Internally, $\mathcal{F}_{\text{FR-CGKA}}$ maintains and dynamically extends the history graph. When $\mathcal{A}$ instructs a party to perform a group operation, the party inputs Send to $\mathcal{F}_{\text{FR-CGKA}}$. The functionality creates a new epoch in its history graph and hands to $\mathcal{A}$ an idealized message. The message is arbitrary, i.e., chosen by the simulator. When $\mathcal{A}$ delivers a message, the party inputs Receive to $\mathcal{F}_{\text{FR-CGKA}}$. On such an input $\mathcal{F}_{\text{FR-CGKA}}$ first asks the simulator to identify the epoch into which the receiver transitions. The simulator can either indicate an existing epoch or instruct $\mathcal{F}_{\text{FR-CGKA}}$ to create a new one. The latter ability should only be used if $\mathcal{A}$ injects a message and, accordingly, epochs created this way are marked as injected. Afterwards, $\mathcal{F}_{\text{FR-CGKA}}$ hands to $\mathcal{A}$ the semantics of the message, computed based on the graph. A corruption in the real world corresponds in the ideal world to $\mathcal{F}_{\text{FR-CGKA}}$ executing the procedure Expose and the simulator computing the corrupted party's state. When $\mathcal{A}$ fetches the group key, the party inputs GetKey to $\mathcal{F}_{\text{FR-CGKA}}$, which outputs a key from the party's epoch. The way keys are chosen is discussed next.

Security guarantees in the ideal world. To formalize confidentiality, $\mathcal{F}_{\text{FR-CGKA}}$ is parameterized by a predicate **conf**, which determines the epochs in the history graph in which confidentiality of the group key is guaranteed. For such a confidential epoch, $\mathcal{F}_{\text{FR-CGKA}}$ chooses a random and independent group key. Otherwise, the simulator chooses an arbitrary key. To formalize authenticity, $\mathcal{F}_{\text{FR-CGKA}}$ is parameterized by **auth**, which determines if authenticity is guaranteed for an epoch and a party. As soon as an injected epoch with authentic parent appears in the history graph, $\mathcal{F}_{\text{FR-CGKA}}$ halts, making the worlds easily distinguishable. Finally, $\mathcal{F}_{\text{FR-CGKA}}$ guarantees consistency by computing the outputs,

[16]To make this section accessible to readers not familiar with UC, we avoid technical details, which sometimes results in inaccuracies. E.g., parties are corrupted by the (dummy) adversary, not $\mathcal{A}$. We hope this doesn't distract readers familiar with UC.

such as the set of group members outputted by a joining party, based on the history graph. This means that the outputs in the real world must be consistent with the graph (and hence also with each other) as well, else, the worlds would be distinguishable.

Observe that the simulator's power to choose epochs into which parties transition and create injected epochs is restricted by the above security guarantees. For example, an injected epoch can only be created if the environment exposed enough states to destroy authenticity. For consistency, $\mathcal{F}_{\text{FR-CGKA}}$ also requires that a party can only transition to a child of its current epoch. Another example is that if a party in the real world outputs a key from a confidential epoch, then the simulator cannot make it transition to an unsafe epoch.

Personalizing messages. saCGKA protocols may require that the mailboxing service personalizes messages before delivering them. In our model, such processing is done by $\mathcal{A}$. It can deliver an honestly processed message, or an arbitrary injected message. The simulator decides if a message is honestly processed, i.e., leads to a non-injected epoch, or is injected, i.e., leads to an injected epoch. Note that this notion has an RCCA flavor. For example, delivering an otherwise honestly generated message but with some semantically insignificant bits modified can still lead the receiver to an honest epoch.

Adaptive corruptions. Our model allows $\mathcal{A}$ to adaptively decide which parties to corrupt, as long as this does not allow it to trivially distinguish the worlds. Specifically, $\mathcal{A}$ can trivially distinguish if a corruption allows it to compute the real group key in an epoch where $\mathcal{F}_{\text{FR-CGKA}}$ already outputted to $\mathcal{A}$ a random key. Our statement quantifies over $\mathcal{A}$'s that do not trivially win.

We note that, in general, there can exist protocols that achieve the following stronger guarantee: Upon a trivial-win corruption, $\mathcal{F}_{\text{FR-CGKA}}$ gives to the simulator the random key it chose and the simulator comes up with a fake state that matches it. However, this requires techniques which typically are expensive and/or require additional assumptions, such as a random oracle programmable by the simulator or a common-reference string. We note that the disadvantage of this is restricted composition in the sense that any composed protocol can only be secure against the class of environments restricted in the same way.

Relation to game-based security. It may be helpful to think about distinguishing between the real and ideal world as a typical security game for saCGKA. The adversary in the game corresponds to the environment $\mathcal{A}$. The adversary's challenge queries correspond to $\mathcal{A}$'s GetKey inputs on behalf of parties in confidential epochs and its reveal-session key queries correspond to $\mathcal{A}$'s GetKey inputs in non-confidential epochs. To disable trivial wins, we require that if the adversary queries a challenge for some epoch, then it cannot corrupt in a way that makes it non-confidential. Apart from the keys in challenge epochs being real or random, the real and ideal world are identical unless one of the following two bad events occurs: First, the adversary breaks consistency, that is, it causes the protocol to output in the real world something different than $\mathcal{F}_{\text{FR-CGKA}}$ in the ideal world. Second, the adversary breaks authenticity, that is, it makes the protocol accept a message that violates the authenticity requirement in the ideal world, making $\mathcal{F}_{\text{FR-CGKA}}$ halt forever. Therefore, distinguishing between the worlds implies breaking consistency, authenticity or confidentiality.

Advantages of simulators. Using a simulator simplifies the notion, because the ideal world does not need to encode parts of the protocol that are not relevant for security. For example, in our model the epochs into which parties transition are arbitrary, as long as security holds. This means that in the ideal world we do not need a protocol function that outputs some unique epoch identifiers. In general, our ideal world is agnostic to the protocol, which is conceptually simple.

Functionality $\mathcal{F}_{\text{AKS}}$

Parameter: key-package generation algorithm `*AKS-kgen`.

```
Initialization
  $\mathsf{SK}[\cdot,\cdot] \leftarrow \bot$

Input GetSK($PK$) from id
  $SK \leftarrow \mathsf{SK}[\mathsf{id}, PK]$
  $\mathsf{SK}[\mathsf{id}, PK] \leftarrow \bot$
  return $SK$

Input (GetPK, id') from id
  $(PK, SK) \leftarrow$ *AKS-kgen()
  $\mathsf{SK}[\mathsf{id}', PK] \leftarrow SK$
  Send $(\mathsf{id}', PK)$ to adv.
  return $PK$
```

Figure 17: The Authenticated Key service Functionality.

B Details of SAIK Components Used by FREEK

τ.root	The root.
v.isroot	True iff $v = \tau$.root.
v.isleaf	True iff v has no children.
v.par	The parent node of v (or $\bot$ if v.isroot).
v.children	If $\neg v$.isleaf: ordered list of v's children.
v.nodeIdx	The node index of v.
v.depth	The length of the path from v to τ.root.
v.pk	An mmPKE encryption key.
v.sk	The corresponding decryption key.
v.spk	If v.isleaf: a signature verification key.
v.ssk	If v.isleaf: the corresponding signing key.
v.unmLvs	The set of indices of the leaves below v whose owner id does not know v.sk.
v.id	If v.isleaf: the id associated with that leaf.

Table 1: Labels of a ratchet-tree τ and its nodes.

γ.grpId	The identifier of the group.
$\gamma.\tau$	The ratchet tree.
γ.leaf	The party's leaf in τ.
γ.treeHash	A hash of the public part of τ.
γ.lastAct	The last modification of the group state and the user who initiated it.
γ.appSec	The current epoch's CGKA key. Exposed to the application layer.
γ.initSec	The next epoch's init secret.
γ.membKey	The next epoch's membership secret for authenticating messages.
γ.parEid	The epoch id of the parent epoch.
γ.confTag	The confirmation tag, which is signed to ensure authenticity.

Table 2: The protocol state of a party id and the helper method for computing the context.

pathSec	The path secrets $s_2, \ldots, s_n$ used to derive the keypairs in each node. Sent via the mmPKE encryption to keep tree invariant intact. s_n is used to derive keys.
joinerSec	Secret sent to new group members. Together with the group context, enables computation of the epSec.

Table 3: Intermediate values computed by the protocol that are not part of the state.

τ.clone()	Returns a copy of τ.
τ.public()	Returns a copy of τ with all labels v.ssk and v.sk set to $\bot$.
τ.roster()	Returns id's of all parties in τ.
τ.leaves()	Returns the list of all leaves in the tree, sorted from left to right.
τ.leafof(id)	Returns the leaf v with v.id = id.
τ.getLeaf()	Returns leftmost v s.t. $\neg v$.inuse(). If no such v exists, adds a new leaf using addLeaf(τ) and returns it.
τ.blankPath(v)	For all $u \in \tau$.directPath(v) calls u.blank().
τ.inSubtree(u, v)	Returns true if u is in v's subtree.
v.inuse()	Returns `false` iff all labels are $\bot$.
v.blank()	Sets all labels of v to $\bot$.
τ.lca(u, v)	Returns the lowest common ancestor of the two leafs.
τ.directPath(v)	Returns the path from v's parent to the root.
τ.mergeLvs(v)	Sets u.unmLvs $\leftarrow \varnothing$ for all $u \in \tau$.directPath(v)
τ.unmerge(v)	Sets u.unmLvs $+\!\!\leftarrow v$ for all u returned by τ.directPath(v)
v.resolution()	If v.inuse, return (v) ++ v.unmLvs. Else if v.isleaf, return (). Else, return v.children[1].resolution() ++ $\cdots$ ++ v.children[n].resolution()
v.resolvent(u)	Returns the ancestor of u in v.resolution() $\setminus$ (v) (or $\bot$ if u is not a descendant of v).

Table 4: Helper methods for a ratchet tree τ and its nodes.

In the current section we recall the components of SAIK [AHKM22b] used by FREEK. The algorithms in Figs. 18 to 20 and Table 4 are copied verbatim from [AHKM22b]. The only difference is that we remove signatures and MACs, as FREEK uses a different mechanism (but we keep signature

SAIK: **Algorithms**

```
Initialization
  if $\mathsf{id} = \mathsf{id}_{\mathsf{creator}}$ then
      $\gamma \leftarrow$ *new-state()
      $\gamma$.grpId, $\gamma$.initSec, $\gamma$.membKey, $\gamma$.appSec $\xleftarrow{\$} \{0,1\}^\kappa$
      $\gamma.\tau \leftarrow$ *new-LBT()
      $\gamma$.leaf $\leftarrow \gamma.\tau$.leaves[0]
      ($\gamma$.leaf.spk, $\gamma$.leaf.ssk) $\leftarrow$ Sig.gen()

Input (Send, act), act $\in$ {'up', 'rem'-$\mathsf{id}_t$, 'add'-$\mathsf{id}_t$}
                                                        from id
  req $\gamma \neq \bot$
  // In case of add, fetch $\mathsf{id}_t$'s keys from AKS (AKS runs *AKS-kgen).
  if act = 'add'-$\mathsf{id}_t$ then
      ($\mathsf{pk}_t, \mathsf{spk}_t, \mathsf{pk}'_t$) $\leftarrow$ query (GetPk, $\mathsf{id}_t$) to $\mathcal{F}_{\text{KS}}$
      act $\leftarrow$ 'add'-$\mathsf{id}_t$-($\mathsf{pk}_t, \mathsf{spk}_t, \mathsf{pk}'_t$)
  // Create the state and secrets for the new epoch.
  try ($\gamma'$, pathSecs, joinerSec) $\leftarrow$ *create-epoch(act)
  // Encrypt the path secrets using the new epoch's ratchet tree. For
  adds, also encrypt the joiner secret.
  if act $\in$ {'up', 'rem'-$\mathsf{id}_t$} then
      $\underline{Ctxt}$ $\leftarrow$ *encrypt($\gamma'$, pathSecs, $\bot, \bot, \bot$)
  else if act = 'add'-$\mathsf{id}_t$-($\mathsf{pk}_t, \mathsf{spk}_t, \mathsf{pk}'_t$) then
      $\underline{Ctxt}$ $\leftarrow$ *encrypt($\gamma'$, pathSecs, $\mathsf{id}_t$, $\mathsf{pk}'_t$, joinerSec)
  ssk $\leftarrow \gamma.\tau$.leafof(id).ssk
  $\gamma \leftarrow \gamma'$
  if act = 'add'-$\mathsf{id}_t$-($\mathsf{pk}_t, \mathsf{spk}_t, \mathsf{pk}'_t$) then
      // Send additional data for $\mathsf{id}_t$.
      welcomeData $\leftarrow$ ($\gamma$.grpId, $\gamma.\tau$.public(), $\mathsf{pk}'_t$)
      return (id, act, $\underline{Ctxt}$, updEKs, welcomeData)
  return (id, act, $\underline{Ctxt}$, updEKs)

Input Key from id
  req $\gamma \neq \bot$
  $k \leftarrow \gamma$.appSec
  $\gamma$.appSec $\leftarrow \bot$
  return $k$

Input (Receive, ($\mathsf{id}_s$, 'removed')) from id
// Receiver is removed.
  spk $\leftarrow \gamma.\tau$.leafof($\mathsf{id}_s$).spk
  $\gamma \leftarrow \bot$
  return ($\mathsf{id}_s$, 'rem'-id)

Input (Receive, ($\mathsf{id}_s$, act, $\underline{ctxt}$, $\underline{\mathsf{updEKs}'}$)) from id
// Receiver is a member.
  try $\gamma' \leftarrow$ *apply-act($\gamma$.clone(), $\mathsf{id}_s$, act)
  try ($\gamma$, confTag) $\leftarrow$ *transition($\gamma'$, $\underline{ctxt}$, $\underline{\mathsf{updEKs}'}$, $\mathsf{id}_s$, act)
  spk $\leftarrow \gamma.\tau$.leafof($\mathsf{id}_s$).spk
  if act = 'add'-$\mathsf{id}_t$-($\mathsf{pk}_t, \mathsf{spk}_t$) then return ($\mathsf{id}_s$, 'add'-$\mathsf{id}_t$)
  else return ($\mathsf{id}_s$, act)

Input (Receive, ($\mathsf{id}_s$, act, $ctxt_1$, $ctxt_2$, welcomeData))) from id
// Receiver joins.
  req $\gamma = \bot$
  parse (grpId, $\tau$, pk') $\leftarrow$ welcomeData
  $\gamma \leftarrow$ *new-state
  ($\gamma$.grpId, $\gamma.\tau$, $\gamma$.lastAct) $\leftarrow$ (grpId, $\tau$, ($\mathsf{id}_s$, 'add'-id))
  $v \leftarrow \gamma.\tau$.leafof(id)
  try (sk, spk, sk') $\leftarrow$ query GetSk(($v$.pk, $v$.spk, pk')) to $\mathcal{F}_{\text{KS}}$
  ($v$.sk, $v$.ssk) $\leftarrow$ (sk, ssk)
  $\gamma \leftarrow$ *set-tree-hash($\gamma$)
  try ($\gamma$, confTag) $\leftarrow$ *get-secrets($\gamma$, sk', $ctxt_1$, $ctxt_2$, $\mathsf{id}_s$)
  return ($\gamma.\tau$.roster(), $\mathsf{id}_s$)
```

SAIK: **Helpers for encryption and key generation for $\mathcal{F}_{\text{AKS}}$**

```
helper *encrypt($\gamma'$, pathSecs, $\mathsf{id}_t$, $\mathsf{pk}'_t$, joinerSec)
  $L \leftarrow$ *rcvrs-of-path-secs($\gamma'.\tau$, id)
  $\vec{U}m, \vec{U}\mathsf{pk} \leftarrow$ ()
  for $j = 1$ to len($L$) do
      $(i, v) \leftarrow L[j]$
      $\vec{U}m$ +$\leftarrow$ pathSecs[$i$]
      if $\mathsf{id}_t \neq \bot \land v = \gamma'.\tau$.leafof($\mathsf{id}_t$) then $\vec{U}\mathsf{pk}$ +$\leftarrow$ $\mathsf{pk}'_t$
      else $\vec{U}\mathsf{pk}$ +$\leftarrow$ $\vec{U}v$.pk
  if $\mathsf{id}_t \neq \bot$ then
      $\vec{U}m$ +$\leftarrow$ joinerSec
      $\vec{\mathsf{pk}}$ +$\leftarrow$ $\mathsf{pk}'_t$
  return $\underline{\text{mmPKE.mmEnc}}$($\vec{\mathsf{pk}}$, $\vec{U}m$)

helper *decrypt-path-secret($\gamma'$, $\mathsf{id}_s$, $ctxt$)
  $v \leftarrow$ lca($\gamma'.\tau$.leafof($\mathsf{id}_s$), $\gamma'$.leaf).resolvent($\gamma'$.leaf)
  return $\underline{\text{mmPKE.mmDec}}$($v$.sk, $ctxt$)

helper *AKS-kgen()
  (pk, sk) $\leftarrow$ $\underline{\text{mmPKE.mmGen}}$()
  (spk, ssk) $\leftarrow$ Sig.gen()
  (pk', sk') $\leftarrow$ $\underline{\text{mmPKE.mmGen}}$()
  return ((pk, spk, pk'), (sk, ssk, sk'))
```

Figure 18: The algorithms of SAIK.

SAIK: **Creating epochs**

```
helper *create-epoch($\gamma$, id, act)
  $\gamma' \leftarrow \gamma$.clone()
  // Apply the action to the tree. Fails if the action is not allowed.
  try $\gamma' \leftarrow$ *apply-act($\gamma'$, id, act)
  // Re-key the direct path.
  directPath $\leftarrow \gamma'.\tau$.directPath($\gamma'$.leaf)
  pathSecs[$*$] $\leftarrow \bot$
  pathSecs[1] $\xleftarrow{\$} \{0,1\}^\kappa$
  for $i = 1$ to len(directPath) $- 1$ do
      $v \leftarrow$ directPath[$i$]
      $r \leftarrow$ HKDF.Exp(pathSecs[$i$], '$node$')
      ($v$.pk, $v$.sk) $\leftarrow$ mmPKE.mmGen($r$)
      pathSecs[$i+1$] $\leftarrow$ HKDF.Exp(pathSec[$i$], '$path$')
  $\gamma'.\tau$.mergeLvs($\gamma'$.leaf)
  // Re-key the leaf.
  ($\gamma'$.leaf.pk, $\gamma'$.leaf.sk) $\leftarrow$ mmPKE.mmGen()
  ($\gamma'$.leaf.spk, $\gamma'$.leaf.ssk) $\leftarrow$ Sig.gen()
  // Set all context variables and then derive epoch secrets.
  $\gamma'$.lastAct $\leftarrow$ (id, act)
  $\gamma' \leftarrow$ *set-tree-hash($\gamma'$)
  ($\gamma'$, joinerSec) $\leftarrow$ *derive-keys($\gamma'$, pathSecs[len(pathSecs)])
  return ($\gamma'$, pathSecs, joinerSec)

helper *apply-act($\gamma'$, id$_s$, act)
  req id$_s \in \gamma'.\tau$.roster()
  if act = '$rem$'-id$_t$ then
      req id$_s \neq$ id$_t \wedge$ id$_t \in \gamma'.\tau$.roster()
      $\gamma'.\tau$.blankPath($\gamma'.\tau$.leafof(id$_t$))
      $\gamma'.\tau$.leafof(id$_t$).blank()
  else if act = '$add$'-id$_t$-(pk$_t$, spk$_t$) then
      req id$_t \notin \gamma'.\tau$.roster()
      $v \leftarrow \gamma'.\tau$.getLeaf()
      ($v$.id, $v$.pk, $v$.spk) $\leftarrow$ (id$_t$, pk$_t$, spk$_t$)
      $\gamma.\tau$.unmerge($v$)

helper *transition($\gamma'$, $ctxt$, updEKs$'$, id$_s$, act)
  // Set keys on the re-keyed path.
  $v_s \leftarrow \gamma'.\tau$.leafof(id$_s$)
  directPath $\leftarrow \gamma'.\tau$.directPath($v_s$)
  ($v_s$.pk, $v_s$.spk) $\leftarrow$ updEKs$'$[1]
  $i \leftarrow 1$
  lca $\leftarrow \gamma'.\tau$.lca($\gamma'$.leaf, $v_s$)
  while directPath[$i$] $\notin$ {lca, $\gamma'.\tau$.root} do
      // If message contains too few ek's, reject it.
      req $i + 1 \leq$ len(updEKs$'$)
      directPath[$i$].pk $\leftarrow$ updEKs$'$[$i+1$]
      $i$++
  // Decrypt the path secret using the updated tree.
  try pathSec $\leftarrow$ *decrypt-path-secret($\gamma'$, id$_s$, $ctxt$)
  while $i <$ len(directPath) do
      $v \leftarrow$ directPath[$i$]
      $r \leftarrow$ HKDF.Exp(pathSecs[$i$], '$node$')
      ($v$.ek, $v$.dk) $\leftarrow$ mmPKE.mmGen($r$)
      pathSec $\leftarrow$ HKDF.Exp(pathSec, '$path$')
      $i$++
  $\gamma'.\tau$.mergeLvs($v_s$)
  // Set all context variables; derive epoch secrets.
  $\gamma'$.lastAct $\leftarrow$ (id$_s$, act)
  $\gamma' \leftarrow$ *set-tree-hash($\gamma'$)
  ($\gamma'$, joinerSec) $\leftarrow$ *derive-keys($\gamma'$, pathSec)
  return $\gamma'$

helper *get-secrets($\gamma'$, dk$'$, $ctxt_1$, $ctxt_2$, id$_s$)
  try pathSec $\leftarrow$ $\underline{\text{mmPKE.mmDec}}$(dk, $ctxt_1$)
  try joinerSec $\leftarrow$ $\underline{\text{mmPKE.mmDec}}$(dk, $ctxt_2$)
  $v \leftarrow \gamma'.\tau$.lca($\gamma'$.leaf, $\gamma'.\tau$.leafof(id$_s$))
  while $v \neq \gamma'.\tau$.root do
      $r \leftarrow$ HKDF.Exp(pathSec, '$node$')
      (pk, $v$.sk) $\leftarrow$ $\underline{\text{mmPKE.mmGen}}$($r$)
      req $v$.pk = pk
      pathSec $\leftarrow$ HKDF.Exp(pathSec, '$path$')
      $v \leftarrow v$.par
  $\gamma' \leftarrow$ *derive-epoch-keys($\gamma'$, joinerSec)
  return $\gamma'$
```

SAIK: **Tree hash**

```
helper *set-tree-hash($\gamma'$)
  $\gamma'$.treeHash $\leftarrow$ *tree-hash($\gamma'.\tau$.root)
  return $\gamma'$

helper *tree-hash($v$)
  if $v$.isleaf then
      return Hash($v$.nodeIdx, $v$.pk, $v$.spk)
  else
      $\ell \leftarrow$ len($v$.children)
      for $i \in [\ell]$ do $h_i \leftarrow$ *tree-hash($v$.children[$i$])
      $h \leftarrow (h_1, \ldots, h_\ell)$
      return Hash($v$.nodeIdx, $v$.pk, $v$.unmLvs, $h$)
```

Figure 19: Additional helper methods for SAIK.

SAIK: Extraction

```
helper $*\mathtt{extract}(\mathsf{id}, \mathsf{act}, \underline{Ctxt}, \mathsf{updEKs}, \sigma, \mathsf{id}_s)$
    $i, j \leftarrow *\mathtt{getExtractionIndices}(\gamma, \mathsf{id})$
    $ctxt \leftarrow \mathsf{mmExt}(C, i)$
    $\mathsf{updEKs}' \leftarrow \mathsf{updEKs}[1 : j]$
    return $\mathsf{id}, \mathsf{act}, ctxt, \mathsf{updEKs}', \sigma$

helper $*\mathtt{getExtractionIndices}(\gamma, \mathsf{id}, \mathsf{id}_s)$
    $v_{\mathsf{lca}} \leftarrow \gamma.\tau.\mathsf{lca}(\mathsf{id}, \mathsf{id}_s)$
    $v_s \leftarrow \gamma.\tau.\mathsf{leafofid}_s$
    $\mathsf{directPath} \leftarrow \gamma.\tau.\mathsf{directPath}(v_s)$
    // Compute number of public keys “under” lca.
    $j \leftarrow \gamma.\tau.\mathsf{leafof}(\mathsf{id}_s).\mathsf{depth} - v.\mathsf{depth}$
    // Count number of encryptions before id’s encryption.
    $k \leftarrow 0$
    for $2 \leq l \leq j$ do
        $k \leftarrow k + \mathtt{len}(\gamma.\tau.\mathsf{resolution}(\mathsf{directPath}[l].\mathsf{children} \setminus \mathsf{directPath}[l-1]))$
    $S \leftarrow \gamma.\tau.\mathsf{resolution}(v_{\mathsf{lca}}.\mathsf{children} \setminus \mathsf{directPath}[j-1])$
    $i \leftarrow 1$
    while $S[i] \cap \gamma.\tau.\mathsf{directPath}(v_R) = \varnothing$ do $i$++
    $i \leftarrow i + k$
    return $i, j$
```

Figure 20: Helper functions for extraction.

key generation).

The authenticated key service (AKS). The AKS is a type of PKI that enables the distribution of key-packages that are used to add new members to the group, without requiring any interaction. It is modeled via the functionality $\mathcal{F}_{\text{AKS}}$ (cf. Fig. 17), which, for simplicity, guarantees that a fresh, authentic, honestly generated key-package for any user is always available.

$\mathcal{F}_{\text{AKS}}$ is parameterized by the key-package generation algorithm, `*AKS-kgen`, and works as follows. `Initialization` initializes an empty array that will be used to store the key-packages. When a party id wants to fetch a key-package of another id′ (via `GetPK`), $\mathcal{F}_{\text{AKS}}$ generates a fresh key-package, stores it, and sends the public part to $\mathcal{A}$ and id. Furthermore, the secret of a key-package can be fetched by the owner id via `GetSK`. Once fetched, the secret key of the key-package is deleted. SAIK is defined in the $\mathcal{F}_{\text{AKS}}$-hybrid model, i.e., its operations make direct calls to the operations provided by $\mathcal{F}_{\text{AKS}}$.

Ratchet trees. SAIK is based on the so-called ratchet trees (RTs), which are left-balanced q-ary tress. For simplicity, we consider $q = 2$, i.e. left-balanced binary trees, LBBT. Informally, an LBBT on n nodes (is defined recursively and) has a maximal full binary tree as its left child, and an LBBT on the remaining nodes, as its right child. In most of the existing CGKA protocols, including SAIK, group members are arranged at the leaves of an RT and all nodes have an associated public-key encryption (PKE) key-pair, except for the root. Leaves nodes can also be associated with signature key-pairs. In the current section we recall the labeling (cf. Table 1) and basic methods (cf. Table 4) over ratchet trees and its nodes, that will assist the presentation of SAIK, while for the formal definitions we refer the reader to [ACDT20, AHKM22b].

SAIK’s state and algorithms. In Table 2, we recall the variables that are contained SAIK’s state (for a single a party). Table 3 lists intermediate secrets computed by the protocol that are not part of the state, but they enable the computation of other secrets.

SAIK’s algorithms are depicted in Figs. 18 and 19. The protocol supports `Initialization`, which enables the initialization of the protocol state, `Send`, which creates new epochs and enables the addition and removal of members, as well as the update of the sender’s state, `Receive`, for processing incoming messages (for new and existing members, as well as removed ones), and `GetKey`, for computing the group key. Only the outputs of `Send` are uploaded to the mailboxing service, and users fetch those messages via `Receive`.

Extraction. The procedure $\texttt{Ext}(C, \mathsf{id}) \to c$ is used by the server to compute the message c that should be delivered to id given the uploaded ciphertext, C. Extraction enables the server to deliver to each id only the part of the ciphertext that should be processed by her. Recall, that the uploaded ciphertext C contains the last group operation, the sender id, a multi-recipient ciphertext and a vector of uploaded public keys, therefore, $\texttt{Ext}(C, \mathsf{id})$ computes id's individual mmPKE ciphertexts and a prefix of the uploaded public keys, which depends on the position of id on the ratchet tress. For further details, see [AHKM22b].