Server-Aided Continuous Group Key Agreement

Anonymous Author(s)

ABSTRACT

Continuous Group Key Agreement (CGKA) – or Group Ratcheting – lies at the heart of a new generation of *scalable* End-to-End secure (E2E) cryptographic multi-party applications. One of the most important (and first deployed) CGKAs is ITK which underpins the IETF's upcoming Messaging Layer Security E2E secure group messaging standard. To scale beyond the group sizes possible with earlier E2E protocols, a central focus of CGKA protocol design is to minimize bandwidth requirements (i.e. communication complexity).

In this work, we advance both the theory and design of CGKA culminating in an extremely bandwidth efficient CGKA. To that end, we first generalize the standard CGKA communication model by introducing *server-aided* CGKA (saCGKA) which generalizes CGKA and more accurately models how most E2E protocols are deployed in the wild. Next, we introduce the SAIK protocol; a modification of ITK, designed for real-world use, that leverages the new capabilities available to an saCGKA to greatly reduce its communication (and computational) complexity in practical concrete terms.

Further, we introduce an intuitive, yet precise, security model for saCGKA. It improves upon existing security models for CGKA in several ways. It more directly captures the intuitive security goals of CGKA. Yet, formally it also relaxes certain requirements allowing us to take advantage of the saCGKA communication model. Finally, it is significantly simpler making it more tractable to work with and easier to build intuition for. As a result, the security proof of SAIK is also simpler and more modular.

Finally, we provide empirical data comparing the (at times, quite dramatically improved) complexity profile of SAIK to state-of-the art CGKAs. For example, in a newly created group with 10K members, to change the group state (e.g. add/remove parties) ITK requires each group member download 1.38MB. However, with SAIK, members download no more than 2.7KB.

CCS CONCEPTS

• **Security and privacy** → **Security protocols**; **Formal security models**;

KEYWORDS

secure group messaging; CGKA; MLS; E2E encryption

1 INTRODUCTION

End-to-end (E2E) secure applications have become one of the most widely used class of cryptographic applications on the internet with billions of daily users. Accordingly, the E2E protocols upon which these applications are built have evolved over several distinct generations, adding functionality and new security guarantees along the way. Modern protocols are generally expected to support features like multi-device accounts, continuous refreshing of secrets and asynchronous communication. Here, *asynchronous* refers to the property that parties can communicate even when they are not simultaneously online. To make this possible, the network provides an (untrusted) mailboxing service for buffering packets until recipients come online.

The growing demand for E2E security motivates increasingly capable E2E protocols; in particular, supporting ever larger groups. For example, in the enterprise setting organizations regularly have natural sub-divisions with far more members than practically supported by today's real-world E2E protocols. Support for large groups opens the door to entirely new applications; especially in the realm of machine-to-machine communication such as in mesh networks and IoT. The desire for large groups is compounded by the fact that many applications treat each device registered to an account as a separate party at the E2E protocol level. For example, a private chat between Alice and Bob who each have a phone and laptop registered to their accounts is actually a 4-party chat from the point of view of the underlying E2E protocol.

Next Generation E2E Protocols. The main reason current protocols (at least those enjoying state-of-the-art security, e.g. post compromise forward security) only support small groups is that their communication complexity grows linearly in the group size. This has imposed natural limits on real-world group sizes (generally at or below 1000 members).

Consequently, a new generation of E2E protocols is being developed both in academia (e.g. [1, 3–5, 7, 8, 25, 31]) and industry [13]. Their primary design goal is to support extremely large groups (e.g. 10s of thousands of users) while still meeting, or exceeding, the security and functionality of today's state-of-the-art deployed E2E protocols. Technically, the new protocols do this by reducing their communication complexity to *logarithmic* in the group size; albeit, only under favorable conditions in the execution. This informal property is sometimes termed the *fair-weather complexity.* protocol.

To date, the most important of these new E2E protocols is the IETF's upcoming secure group messaging (SGM) standard called the *Messaging Layer Security* (MLS) protocol. MLS is in the final stages of standardization and its core components are already seeing initial deployment [24].

Continuous Group Key Agreement. To the best of our knowledge, all next gen. E2E protocols share the following basic design paradigm. At their core lies a *Continuous Group Key Agreement* (CGKA) protocol; a generalization to the group setting of the *Continuous Key Agreement* 2-party primitive [4, 28] underlying the Double Ratchet.

Intuitively, a CGKA protocol provides *E2E secure group management* for dynamic groups, i.e., groups whose properties may change mid-session. By properties we mean things like the set of members currently in the group, their attributes, the group name, the set of moderators, etc. Any change to a group's properties initiates a fresh *epoch* in the session. A CGKA protocol ensures all group members in an epoch agree on the group's current properties. Members will only transition to the same next epoch if they agree on which properties were changed and by whom. Each epoch is equipped with its own symmetric *epoch key* known to all epoch members but indistinguishable from random to anyone else. Higher-level protocols typically (deterministically) expand the epoch key into a complete key schedule which in turn can be used to, say, protect application data sent between members (e.g. messages or VoiP data).

MLS too, is (implicitly) based on a CGKA, originally dubbed *TreeKEM* [16]. Since its inception, TreeKEM has undergone several substantial revisions [10, 11] before reaching its current form [8, 12]. For clarity, we refer to its current version at the time of this writing as *Insider-Secure TreeKEM* (ITK) (using the terminology of [8] where that version was analyzed). ITK has already seen its first real world deployment as a core component of Cisco's Webex conferencing protocol [24].

Why Consider CGKA? CGKA is interesting because of the following two observations. First, CGKA seems to be the minimal functionality encapsulating almost all of the cryptographic challenges inherent to building next generation E2E protocols. Second, building typical higher-level E2E applications (e.g. SGM or conference calling) from a CGKA can be done via relatively generic, and comparatively straightforward mechanisms. Moreover, the resulting application directly inherits many of its key properties from the underlying CGKA; notably their security guarantees and their communication and computational complexities. In this regard, CGKA is to, say, SGM what a KEM is to hybrid PKE. For the case of SGM, this intuitive paradigm and the relationship between properties of the CGKA and resulting SGM was made formal in [6]. In particular, that work abstracts and generalizes MLS's construction from ITK.

1.1 Our Contributions

This work makes progress on the central challenge in CGKA protocol design: reducing communication complexity so as to support larger groups (without compromising on security or functionality).

Server-Aided CGKA. We begin by revisiting one of the most basic assumptions about CGKA in prior work; namely that participants communicate via an insecure broadcast channel. Instead, we note that in almost all modern deployments of E2E protocols parties actually communicate via an untrusted mailboxing service implemented using an (often highly scalable) *server*. In light of this, we modify the standard communication model to make the server explicit. Correspondingly we define a generalization of CGKA called *server-aided* CGKA (saCGKA).An saCGKA protocol includes an *Extract* procedure run by the server to convert a "full packet" uploaded by a sender into an individualized "sub-packet" for a particular recipient. CGKA corresponds to the special case where the full and individual packets are the same. Intuitively, the server remains untrusted and security should hold no matter what it does. However, should it choose to follow the Extract procedure, the saCGKA protocol ensures correctness and availability.

Security for CGKA. We define a new security notion for saCGKA capturing the same intuitive guarantees as those shown for ITK [8] for example. Like other notions based on the history graph paradigm of [6], our notion is parameterized by *safety* predicates that together decide the security of a target epoch key in a given execution.

However, at a technical level our notion departs significantly from past ones. Essentially, it relaxes the requirement that group members in an epoch agree on and authenticate the *history of network traffic* leading to the epoch. Instead, the new notion "only" ensures they agree on and authenticate the *semantics* of the history; i.e. the "meaning" of the traffic rather than exact packet contents. This has several interesting consequences. First, it more directly captures our intuitive security goals. E.g. it avoids subtle questions about what intuition is really captured when, say, an AEAD ciphertext in a packet can be decrypted to different plaintexts using different keys.[1] Second, the relaxation creates wiggle room we can use to prove security despite group members no longer having the same view of network traffic. Finally, it allows us to relax the security of the encryption scheme used in our construction from CCA to *replayable CCA* (RCCA)[22].[2]

The new saCGKA security notion is significantly simpler (though just as precise) compared to past ones. Past notions have been criticised for being all but inaccessible to non-domain experts due to their complexity. In an effort to improve this, our new notion omits/simplifies various security features of a CGKA as long as A) they can be formalized using known techniques and B) they can be easily achieved by known, practical and straightforward extensions of a generic CGKA protocol (including SAIK) satisfying our notion. Thus we obtain a definition focused on the basic properties of an (sa)CGKA with the idea that a protocol satisfying our notion can easily be extended to a "full-fledged" (sa)CGKA using standard techniques.

The SAIK *Protocol.* Next, we introduce a new saCGKA protocol called *Server-Aided* ITK (SAIK), designed for real-world use. For example, it relies exclusively on standard cryptographic primitives and can be implemented using the API of various off-the-shelf cryptographic libraries. To obtain SAIK, we start with ITK and make the following modifications.

Multi-message multi-recipient PKE. First, we replace ITK's use of standard (CCA secure) PKE with multi-message multi-recipient PKE (mmPKE) [36]. Directly constructing mmPKE can result in a significantly more efficient scheme than produced by parallel composition of standard PKE schemes (both in terms of ciphertext sizes and computation cost of encryption).

We introduce a new security notion for mmPKE, more aligned with the needs of (the security targeted by) SAIK. It both strengthens and weakens past notions: On the one hand, proving SAIK secure demands that we equip the mmPKE adversary of [36] with adaptive key compromise capabilities. On the other hand, thanks to the relaxation to semantic agreement, we "only" require RCCA security rather than full-blown CCA used in previous' works [7, 8].

We prove the mmPKE construction of [36] satisfies our new notion based on a form of gap Diffie-Hellman assumption, the same as in [36]. The reduction is tight in that the security loss is independent of the number of parties (i.e. key pairs) in the execution (although it does depend on the number of corrupted key pairs). Moreover, we extend the proof to capture mmPKE constructions based on "nominal groups" [2]. Nominal groups abstract the algebraic structure over bit-strings implicit to the X25519 and X448 scalar multiplication functions and corresponding twisted Edwards

[1] This can happen for widely used AEADs like AES-GCM [27].

[2] This makes sense as RCCA was designed to relax the "syntactic non-malleability" of CCA to a form of "semantic non-malleability".

curves.[35]. In practical terms, this means our proofs also apply to instantiations of [36] that are based on the X25519 and X448 functions.

Authentication and Agreement. Second, we modify the mechanisms used by ITK to ensure members transitioning to a new epoch authenticate who is making changes to the group's properties when a new epoch is announced. Rather than sign the full packet like in ITK, a sender in SAIK only signs a small tag common to view of all receivers. This reduces the amount of data receivers must download to verify the signature. We also modify how changes to the group's properties are incorporated in the derivation of the new epoch key. Rather include a hash of the full packet representing the change, SAIK uses a hash of a symbolic representation of the change. Proving this secure leverages the new wiggle room in our CGKA security created by introducing semantic agreement.

Performance Evaluation. Finally, we provide empirical data comparing the communication complexity for senders and receivers running various instantiations of SAIK and ITK for a variety of execution profiles. Our results show that for senders SAIK reduces communication complexity (and halves the number of public key operations) compared to ITK. Specifically, for 10K parties, sender communication complexity decreases from 4.4KB down to 3.6KB in the best case and from 1.5MB to 0.77MB in the worst case. Meanwhile for receivers the communication complexity goes from anywhere between logarithmic and even linear in the group size of ITK down to at most logarithmic for SAIK. Concretely, in a freshly created group with 10K parties a receiver in a ITK session needs to download 1.38MB to transition into a new epoch while the same receiver in SAIK downloads no more than 2.7KB.

Outline of the paper. The paper is structured as follows. Sec. 2 (and App. A) covers basic preliminaries. Sec. 3 focuses on mmPKE while Sec. 4 describes the new security model for saCGKA. Sec. 5 describes the SAIK protocol. The intuition for its security (i.e. its safety predicates) are in Sec. 6. Finally, Sec. 7 contains empirical evaluation and comparison of SAIK to previous constructions. The formal specification of the (sa)CGKA security model is in App. D. **??** contains the exact safety predicate, concrete security statement and security proof for SAIK.

1.2 Related Work

Next generation CGKA protocols. The study of next generation CGKA protocols for very large groups was initiated by Cohn-Gorden et al. in [25]. This was soon followed by the first version of TreeKEM [37] which evolved to add stronger security [10, 37, 39] and more flexible functionality [11] culminating in its current form ITK [8] reflected in the current draft of the MLS RFC [12].

Reducing the communication complexity of TreeKEM and its descendants is not a new goal. *Tainted TreeKEM* [3] exhibits an alternate complexity profile optimized for a setting where the group is managed by a small set of moderators. Recently, [1] introduced new techniques for 'multi-group" CGKAs (i.e. CGKAs that explicitly accommodate multiple, possibly intersecting, groups) with better complexity than obtained by running a "single-group" CGKA for each group. Other work has focused on stronger security notions for CGKA both in theory [7] and with an eye towards practice [5, 8]. Supporting more concurrency has also been a topic of focus as witnessed by the protocols in [11, 18, 40]. Recently [30] present CGKA with novel membership hiding properties.

Cryptographic models of CGKA security. Defining CGKA security in a simple yet meaningful way has proven to be a serious challenge. Many notions fall short in at least one of the two following senses. Either they do not capture key guarantees desired (and designed for) by practitioners (such as providing guarantees to newly joined members) or they place unrealistic constraints on the adversary. Above all, they do not consider fully active adversaries. For instance, in [3], the adversary is not allowed to modify packets while in [5, 6], new packets can be injected but only when authenticity can be guaranteed despite past corruptions (thus limiting what is captured about how session's regain security after corruptions). Meanwhile, the work of [19] permits a large class of active attacks but only in the context of the key derivation process of ITK. So while their adversaries can arbitrarily modify secrets in an honest party's key derivation procedure, they can not deliver arbitrary packets to honest parties. This is a significant limitation, e.g., it does not capture adversaries that deliver packets with ciphertexts for which they do not know the plaintexts.

Indeed, a good indication that such simplifications can be problematic can be found in [8]. They present an attack on TreeKEM (that can easily be easily adapted to the CGKAs in the above works except for [19]) which uses honest group members as decryption oracles to clearly violate the intuitive security expected of a CGKA. Yet, each of the above works (except for [19]) proves security of their CGKA using only IND-CPA secure encryption.

In contrast to the above works, [7] aimed to capture the full capabilities a realistic adversary might have. Thus, they model a fully active adversary that can leak parties local states at will and even set their random coins. In [8] this setting is extended to capture *insider* security. That is adversaries which can additionally corrupt the PKI. This captures the standard design criterion for deployed E2E applications that key servers are *not* considered trusted third parties. Unfortunately, this level of real-world accuracy has resulted in a (probably somewhat inherently) complicated model.

Symbolic models of CGKA security. Complementing the above line of work, several versions of TreeKEM have been analyzed using a symbolic approach and automated provers [17]. Their models consider fully active attackers and capture relatively wide ranging security properties which the authors are able to convincingly tackle by using automated proofs.

Concurrent Work. Concurrently and independently to this work [31] present an interesting saCGKA and corresponding adaptation of the insider security notion of [8]. Like our work, they consider a server that extracts receiver specific sub-packets from a senders full packet. However, their security notion still enforces representational rather than semantic agreement on the transcript history. They use a novel multi-recipient (but not multi-message) PKE which is shown to be post-quantum secure (though their CGKA is only proven classically secure).

The efficiency of their protocol is incomparable to ours. While with SAIK a receiver's sub-packet contains between 1 to $\log(n)$ public keys (for groups of size n) in [31] a receiver never needs to download more than a single public key. However, both the computational and communication complexity for senders is *always* linear in n.[3] Meanwhile SAIK enjoys logarithmic fair-weather complexity. That is, sender complexity may vary between linear down logarithmic (or even constant). In all but the most pathological executions, other than when a group is freshly created we expect complexity to be roughly logarithmic for the vast majority of the execution.

mmPKE. mmPKE was first proposed by Kurosawa [34] though their security model was flawed as pointed out and fixed by Bellare et.al [14, 15]. Yet, those works too lacked generality as they demanded malicious receivers know a secret key for their public key. This restriction was lifted by Poettering et.al. in [36] who show that well-known PKE schemes such as ElGamal[29] and Cramer-Shoup [26] are secure even when reusing coins across ciphertexts. Indeed, reusing coins this way can also reduce the computational complexity of encapsulation and the size of ciphertexts for KEMs as shown in the Multi-Recipient KEM (mKEM) of [23, 33, 38] for example. All previous security notions (for mmPKE and mKEM) allow an adversary to provide malicious keys (with or without knowing corresponding secret keys), but none allow for adaptive corruption of honest keys, which is necessary for ITK's security against adaptive adversaries.

2 NOTATION

For $n \in \mathbb{Z}$, we define $[n] := \{1, 2, \ldots, n\}$. We write $x \xleftarrow{\$} X$ for sampling an element x uniformly at random from a (finite) set X as well as for the output of a randomized algorithm, i.e. $x \xleftarrow{\$} A(y)$ denotes the output of the probabilistic algorithm A on input y using fresh random coins. For a deterministic algorithm A, we write $x = A(y)$. Adding an element y to a set Y is denoted by $Y \xleftarrow{+} y$ and appending an entry z to a list L is written as $Z \xleftarrow{+\!\!+} z$. Appending a whole list L_2 to a list L_1 is denoted by $L_1 +\!\!+ L_2$. For a vector $\vec{x}$, we denote its length as $|\vec{x}|$ and $\vec{x}[i]$ denotes the i-th element of $\vec{x}$ for $i \in [|\vec{x}|]$. Note that we use vectors as in programming, i.e. we don't require any algebraic structure on them. For clarity, we use `len` to denote the length of collections.

3 MULTI-MESSAGE MULTI-RECIPIENT PKE

We first recall the syntax of mmPKE from [15]. At a high level, mmPKE is standard encryption that supports batching a number of encryption operations together, in order to improve efficiency.[4]

*Definition 3.1 (*mmPKE*).* A Multi-Message Multi-Receiver Public Key Encryption (mmPKE) scheme mmPKE = (KG, Enc, Dec, Ext) consists of the following four algorithms:

- $\mathsf{KG} \xrightarrow{\$} (\mathsf{ek}, \mathsf{dk})$: Generates a new key pair.
- $\mathsf{Enc}(\vec{\mathsf{ek}}, \vec{m}) \xrightarrow{\$} C$: On input of a vector of public keys $\vec{\mathsf{ek}}$ and a vector of messages $\vec{m}$ of the same length, outputs a *multi-recipient ciphertext* C encrypting each message in $\vec{m}$ to the corresponding key in $\vec{\mathsf{ek}}$.
- $\mathsf{Ext}(C, i) \to c_i$: A deterministic function. On input a multi-recipient ciphertext C and a position index i, outputs an *individual ciphertext* c_i for the i-th recipient.
- $\mathsf{Dec}(\mathsf{dk}, c) \to m \vee \perp$: On input of an individual ciphertext c and a secret key dk, outputs either the decrypted message m or, in case decryption fails, $\perp$.

3.1 Security with Adaptive Corruptions

Our security notion for mmPKE requires indistinguishability in the presence of active adversaries who can adaptively corrupt secret keys of recipients. The notion builds upon the (strengthened) IND-CCA security of mmPKE from [36], but there are two important differences: First, [36] does not consider corruptions. Second, instead of CCA, we define the slightly weaker notion of Replayable CCA (RCCA). Roughly, RCCA [22] is the same as CCA except modifying a ciphertext so that it encrypts the exact same message is not considered an attack. RCCA security is implied by CCA security.

We note that an almost identical security definition was presented in parallel by Hashimoto et.al.[31]. However, they only consider multi-recipient PKE (mPKE), where all recipients receive the same message.

Our security notion is called mmIND-RCCA and is similar to RCCA security of regular encryption in the multi-user setting. The main difference is that the challenge ciphertext is computed by encrypting one of two *vectors* of messages $\vec{m}_0^*$ and $\vec{m}_1^*$ under a *vector* of public keys $\vec{\mathsf{ek}}^*$. The vector $\vec{\mathsf{ek}}^*$ is chosen by the adversary and can contain keys generated by the challenger as well as arbitrary keys. The adversary also gets access to standard decrypt and corrupt oracles for each recipient. To disable trivial wins, we require that $\vec{m}_0^*$ and $\vec{m}_1^*$ have equal lengths and that if the i-th key in $\vec{\mathsf{ek}}^*$ is corrupted, then the i-th components of $\vec{m}_0^*$ and $\vec{m}_1^*$ are identical. Moreover, we require that the for each i, the lengths of the i-th components of $\vec{m}_0^*$ and $\vec{m}_1^*$ are the same.

The last requirement means that a secure mmPKE scheme may leak the lengths of components of encrypted vectors. We note that SAIK is also secure when instantiated with an mmPKE scheme that leaks more (see App. G), e.g. whether two messages in a vector are the same. The formal definitions are in App. A.3.

3.2 Construction

The mmPKE of [36] is straightforward. It requires a data encapsulation scheme DEM, a hash H and a group $\mathbb{G}$ of prime order p generated by g.[5] Recall that ElGamal encryption of m to public key g^x requires sampling coins r to obtain ciphertext $(g^r, \mathsf{DEM}(k_m, m))$ where $k_m = H(g^{rx}, g^x)$. The mmPKE variant reuses coins r from the first ElGamal ciphertext to encrypt all subsequent plaintexts. Thus, the final ciphertext has the form $(g^r, \mathsf{DEM}(k_1, m_1), \mathsf{DEM}(k_2, m_2), \ldots)$ where $k_i = H(g^{rx_i}, g^{x_i})$ for all i. We call the construction $\mathsf{DH\text{-}mmPKE}[\mathbb{G}, g, p, \mathsf{DEM}, H]$ and give a formal description in Fig. 8 in App. C.

Optimizing for Short Messages. Normally, when messages m can have arbitrary size, a sensible mmPKE would use a KEM\DEM style construction to avoid having to re-encrypt m multiple times. In

[3]In fact, this complexity is roughly a half that of a CGKA construction built from a network of 2-party double-ratchet channels similar to the sender keys protocol used by WhatsApp.

[4]The majority of works on mmPKE uses a different syntax, where there is no Ext and instead Enc outputs a vector of individual ciphertexts. Since Ext is deterministic, the syntaxes are equivalent.

[5]In SAIK we can instantiate DEM with an off-the-shelf AEAD such as AES-GCM and H with HKDF.

other words, for each m in the encrypted vector, we choose a fresh key k'_m for an AEAD and encrypt m with k'_m. Then use the mmPKE of [36] to encrypt k'_m to each public key receiving m. However, since the secrets encrypted in SAIK have the same length as AEAD keys, in our case it is more efficient to encrypt the secrets directly.

Tight security bound. In App. C, we prove the following upper bound on the advantage of any adversary against the mmPKE from [36]. Our bound is tighter than the bound that follows from the straightforward adaptation of the bound from [36] (i.e. using the hybrid argument and guessing the uncorrupted key). In particular, that bound would depend (linearly) on the total number of public keys, which may get very large. In contrast, our bound depends only on the number of corrupted keys and the length of encrypted vectors.

THEOREM 3.2. *Let $\mathbb{G}$ be a group of prime order p with generator g, let* DEM *be a data encapsulation mechanism and let* mmPKE = DH-mmPKE$[\mathbb{G}, g, p, \mathsf{DEM}, H]$. *For any adversary $\mathcal{A}$ and any $N \in \mathbb{N}$, there exist adversaries $\mathcal{B}_1$ and $\mathcal{B}_2$ with runtime roughly the same as $\mathcal{A}$'s s.t.*

$$\mathrm{Adv}^{\mathsf{mmIND\text{-}RCCA}}_{\mathsf{mmPKE},N}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{IND\text{-}RCCA}}_{\mathsf{DEM}}(\mathcal{B}_2) + 2n \cdot (e^2 q_c \mathrm{Adv}^{\mathsf{DSSDH}}_{(\mathbb{G},g,p)}(\mathcal{B}_1) + \frac{q_{d_1}}{p} + \frac{q_h}{p}),$$

where H is modeled as a random oracle, e is the Euler number, n is the length of the challenge vector, and q_{d_1}, q_c and q_h are the number of queries to the decrypt and corrupt oracles and the random oracle, respectively.

Remark 1. Some practical applications of Diffie-Hellman, most notably CURVE25519 and CURVE448 [35], implement a Diffie-Hellman operation that is not exponentiation in a prime-order group. Such operations can be formalized as so-called nominal groups [2]. In App. B, we generalize and prove Theorem 3.2 for nominal groups. In particular, this means that DH-mmPKE is secure if instantiated with CURVE25519 and CURVE448.

4 SERVER-AIDED CGKA

In this section, we first explain the saCGKA syntax, i.e., the interface exposed by saCGKA protocols to higher-level applications. Then, we give intuitive security properties saCGKA protocols should provide and an overview of our saCGKA security model. For details, see App. D. Finally, we highlight the additional flexibility provided by semantic agreement of saCGKA and list simplifications it makes compared with previous works on active CGKA security [7, 8, 31].

4.1 Syntax

A saCGKA protocol allows a dynamic group of parties to agree on a continuous sequence of symmetric group keys. An execution of a saCGKA protocol proceeds in *epochs*. During each epoch, a fixed set of current group members shares a single group key. A group member can modify the group state, that is, create a new epoch, by sending a single message to the mailboxing service. Afterwards, each group member can download a possibly personalized message and, if they accept it, transition to the new epoch. Three types of group modifications are supported: adding a member, removing a member and updating, i.e., refreshing the group key.

4.2 Intuitive Security Properties

saCGKA protocols are designed for the setting with *active adversaries* who fully control the mailboxing service and repeatedly expose secret states of parties. Note that, unless some additional uncorruptible resources such as a trusted signing device are assumed, the above adversary subsumes the typical notion of malicious insiders (or actively corrupted parties in MPC).

To talk about security of saCGKA, we use the language of *history graphs* introduced in [6]. A history graph is a symbolic representation of group's evolution. Nodes represent epochs and directed edges represent group modifications. For example, when Alice in epoch E wants to add Bob, she creates an epoch E' with an edge from E to E'. The graph also stores information about parties' current epochs, the adversary's actions, etc.

In a perfect execution, the history graph would be a chain. However, even for benign reasons, this may not be the case. For example, if two parties simultaneously create epochs, then a fork in the graph is created. Moreover, an active adversary can deliver different messages to different parties, causing them to follow different branches. Further, it can trick parties into joining fake groups it created by injecting invitation messages. Epochs in fake groups form what we call *detached trees.* So, in full generality, the graph is a directed forest. Using history graphs we can list intuitive security properties of saCGKA.

Consistency 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 consistency is agreement on the transcript: any two parties in one epoch reached it by executing the same sequence of group modifications since the latter one joined.

Confidentiality An epoch is confidential if the adversary has no information about its group key. Corruptions may destroy confidentiality in certain epochs. saCGKA security is parameterized by a *confidentiality predicate* which identifies confidential epochs in an execution.

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

The confidentiality and authenticity predicates generalize forward-secrecy and post-compromise security.

4.3 Authenticated Key Service (AKS)

Most CGKA protocols, including ITK and SAIK, rely on a type of PKI called here the Authenticated Key Service (AKS). The AKS authentically distributes so-called one-time key packages (also called key bundles or pre-keys) used to add new members to the group without interacting with them. For simplicity, we use an idealized AKS which guarantees that a fresh, authentic, honestly generated key package of any user is always available.

4.4 Formal Model Intuition

We define security of saCGKA protocols in the UC framework. That is, a saCGKA protocol is secure if no environment $\mathcal{A}$ can distinguish between the real world where it interacts with parties executing the protocol and the ideal world where it interacts with the ideal saCGKA functionality and a simulator. Readers familiar with game-based security should think of $\mathcal{A}$ as the adversary (see also App. A.1 for some additional discussion).

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 a message updates its state (or creates it in case of new members) and hands to $\mathcal{A}$ the semantic of the 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.[6]

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, $\mathcal{F}_{\text{CGKA}}$. Internally, $\mathcal{F}_{\text{CGKA}}$ maintains and dynamically extends a history graph. When $\mathcal{A}$ instructs a party to perform a group operation, the party inputs Send to $\mathcal{F}_{\text{CGKA}}$. The functionality creates a new epoch in its history graph and hands to $\mathcal{A}$ an idealized message. The message is chosen by an arbitrary simulator, which means that it is arbitrary. When $\mathcal{A}$ delivers a message, the party inputs Receive to $\mathcal{F}_{\text{CGKA}}$. On such an input $\mathcal{F}_{\text{CGKA}}$ first asks the simulator to identify the epoch into which the receiver transitions. The simulator can either indicate an existing epoch or instruct $\mathcal{F}_{\text{CGKA}}$ 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, $\mathcal{F}_{\text{CGKA}}$ 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 $\mathcal{F}_{\text{CGKA}}$ 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 $\mathcal{F}_{\text{CGKA}}$, 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, $\mathcal{F}_{\text{CGKA}}$ is parameterized by a predicate **confidential**, which determines the epochs in the history graph in which confidentiality of the group key is guaranteed. For such a confidential epoch, $\mathcal{F}_{\text{CGKA}}$ chooses a random and independent group key. Otherwise, the simulator chooses an arbitrary key. To formalize authenticity, $\mathcal{F}_{\text{CGKA}}$ is parameterized by **authentic**, 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, $\mathcal{F}_{\text{CGKA}}$ halts, making the worlds easily distinguishable. Finally, $\mathcal{F}_{\text{CGKA}}$ guarantees consistency by computing the outputs, 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, $\mathcal{F}_{\text{CGKA}}$ 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 safe 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 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 $\mathcal{F}_{\text{CGKA}}$ 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, $\mathcal{F}_{\text{CGKA}}$ 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 $\mathcal{F}_{\text{CGKA}}$ 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 $\mathcal{F}_{\text{CGKA}}$ halt forever. Therefore, distinguishing between the worlds implies breaking consistency, authenticity or confidentiality.

[6]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.

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. Our ideal world is agnostic to the protocol, which is conceptually simple.

4.5 Semantic Agreement

An important difference between our model and those of [7, 8, 31] is that in [7, 8, 31] epochs are (uniquely) identified by messages creating them. This is problematic for saCGKA, because different receivers transition to a given epoch using different messages. Crucially, this means an injected message cannot be used to identify the injected epoch into which its receiver transitions. We deal with this in a clean way by allowing the simulator to identify epochs. That is, epoch identifiers are arbitrary as long as consistency, authenticity and confidentiality hold.

The work [31], which proposes a new CGKA where, similar to SAIK, receivers get personalized packets, encountered the same problem with the existing models [7, 8]. In their new model, filtered CGKA (fCGKA), an epoch is identified by the sequence of *packet headers* leading to it. The header is a part of the uploaded packet that is downloaded by all receivers. A protocol can be secure according to the fCGKA model only if the header it defines has the properties of a cryptographic commitment to the semantics of the packet.

saCGKA generalizes fCGKA (and [7, 8]) and provides additional flexibility. For instance, it enables CGKAs which, like SAIK, assume PKE with the weaker RCCA security, while fCGKA still requires the stronger notion of CCA. We believe that in the future more CGKA protocols will take advantage of saCGKA's flexibility. For example, one may consider using a different packet-authenticator for each receiver with the goal of providing some level of unlinkability – an adversary seeing only packets downloaded by participants cannot tell if they are in the same epoch (or group) or not.

4.6 Simplifications

In order to make the security notion tractable, we made the following simplifications compared to the models of [7, 8, 31].

Immediate transition In our model, a party performing a group operation immediately transitions to the created epoch. In reality, a party would only send the message creating the epoch and wait for an ACK from the mailboxing service before transitioning. If it receives a different message before the ACK, it transitions to that epoch instead. This mitigates the problem that if many parties send at once then they end up in parallel epochs and cannot communicate.

A protocol *Prot* implementing immediate transition can be transformed in a black-box manner into a protocol *Prot'* that waits for ACK as follows: To perform a group operation, *Prot'* creates a copy of the current state of *Prot* and runs *Prot* to obtain the provisional updated state and the message. The message is sent and all provisional states are kept in a list. If some message is ACK'ed, the corresponding provisional state becomes the current one, and if another message is receiver, it is processed using the current state. In any case, all provisional states are cleared upon transition.

Simplified PKI The models of [8, 31] consider a realistic implementation of the AKS where parties generate key packages themselves and upload them to an untrusted server, authenticated with long-term so-called identity keys. These long-term keys are authenticated via a PKI which allows the adversary to leak registered keys and even to register their own arbitrary keys on behalf of any participant. The works [8, 31] define fine-grained security in this setting, i.e., their security predicates take into account which PKI keys delivered to parties were corrupted.

In contrast, our model avoids the complexity of keeping track of the PKI keys in $\mathcal{F}_{\text{CGKA}}$, at the cost of more coarse-grained guarantees. For example, it no longer captures the (subtle) security guarantees provided by (the tree-signing of) ITK to parties invited to fake groups created by the adversary (tree signing trivially works for SAIK). We stress that our model does capture most active attacks, e.g. injecting valid-looking packets that add parties with arbitrary injected key packages.

Deleting group keys To build a secure messaging protocol on top (sa)CGKA, it is important that (sa)CGKA removes from its state all information about the group key K immediately after outputting it. The reason is that the messaging protocol will symetrically ratchet K forward for FS. If the initial K was kept in the (sa)CGKA state upon corruption, the adversary could recompute all symmetric ratchets in the current epoch, breaking FS. Our $\mathcal{F}_{\text{CGKA}}$ does not enforce that K is deleted, in order to avoid additional bookkeeping. All natural protocols, including SAIK, can trivially delete K, as it is stored as a separate variable that is computationally independent of the rest of the state.

No randomness corruptions Our model does not capture attacks where the adversary exposes or modifies the randomness used by the protocol. Capturing such attacks for (sa)CGKA causes a significant headache when defining the formal security notion. For instance, the model needs to special-case scenarios where the adversary leaks the state of a party A, uses it with randomness r to compute and inject a message to a party B, and then makes A use r to re-compute the injected message.

One can easily adapt the special-casing of [7, 8, 31] to our model. We chose not to do this for simplicity and because well-designed protocols, including ITK and SAIK, naturally have protections against bad randomness.()Looking ahead, these protocols mix a fresh "commit" secret for the new epoch with the "init" secret from the old epoch, which mitigates sampling the fresh secret with bad randomness.) Therefore, capturing the additional attack vector typically does not fundamentally improve the analysis.

Simplified syntax To improve efficiency, ITK and CmPKE of [31] use the so-called propose-commit syntax, originally proposed by MLS. This means that parties first send messages that *propose* adding or removing other members, or updating their own keys. This does not affect the group state immediately. Rather, a party can collect a list of proposals and send a *commit* message which applies the proposed changes and creates a new epoch. The advantage of this is avoiding the expensive operation of epoch creation after every group modification (modifications typically come in batches; for instance, lots of members are added immediately after group creation).

Unfortunately, using this syntax requires lots of additional bookkeeping from $\mathcal{F}_{\text{CGKA}}$, such as keeping track of two types of history-graph nodes, one for proposals and one for commits (see [8, 31]). Most protocols based on ITK, such as SAIK, can be easily adapted to the propose-commit syntax and benefit from the efficiency gain. The change is minimal and security proofs are clearly not affected. We elaborate in **??**.

No correctness guarantees Our model does not capture correctness, i.e., the simulator can always make a party reject a message. Therefore, a protocol that does nothing is secure according to the notion. This greatly simplifies the definition and the fact that a protocol is correct typically easily follows by inspection (which is often the core argument in the proof of correctness). This means that the protocol used by the mailboxing service to extract personalized packets is not part of the security notion – a fully untrusted service may anyway deliver arbitrary packets. We note that the above protocol is still a part of saCGKA.

5 THE SAIK PROTOCOL

SAIK inherits most of its mechanisms from ITK, the CGKA of MLS. We briefly recall ITK in Sec. 5.1. Readers familiar with ITK can jump directly to Sec. 5.2 which gives intuition how SAIK improves on ITK. The detailed description of SAIK can be found in App. F.

5.1 Intuition for the ITK Protocol

Ratchet trees. The operation of ITK relies on a data structure called ratchet trees. A ratchet tree τ is a tree where leaves are assigned to group members, each storing its owner's identity and signature key pair. Moreover, most non-root nodes in τ, store encryption key pairs. Nodes without a key pair are called *blank*.

ITK maintains the following *tree invariant*: *Each member knows the secret keys of the nodes on the path from their leaf to the root, and only those, as well as all public keys in τ.* This allows to efficiently encrypt messages to subgroups: If a node v is not blank, then a message m can be encrypted to all members in the subtree of a node v by encrypting it under v's public key. If v is blank, then the same can be achieved by encrypting m under each key in v's *resolution*, i.e., the minimal set of non-blank nodes covering all leaves in v's subtree.

Ratchet tree evolution. Each group modification corresponds to a modification of the ratchet tree τ. Most importantly, an update performed by a member id corresponds to refreshing all key pairs with secret keys known to id, i.e., those in the nodes on the path from id's leaf to the root. id generates the new key pairs and, to maintain the tree invariant, communicates the secret keys to some group members. This is done efficiently as follows.

(1) Let $v_1, \dots, v_n$ denote the nodes on the path from id's leaf v_1 to the root v_n. id generates a sequence of *path secrets* $s_2, \dots, s_n$: s_2 is a random bitstring, $s_{i+1} = \mathsf{Hash}(s_i, \text{'path'})$.
(2) id generates a fresh key pair for v_1. For each $i \in [2, n-1]$, the new key pair of v_i is computed by running the key generation with randomness $\mathsf{Hash}(s_i, \text{'rand'})$. The last secret s_n will be used in the key schedule, described soon.
(3) id encrypts each s_{i+1} to the sibling of v_i. This allows members in the subtree of v_i (and only those) to derive $s_{i+1}, \dots, s_n$.

Each add and remove is immediately followed by an implicit update. Removing a member id_t corresponds to removing all keys known to it, i.e., blanking all nodes on the path from its leaf to the root. Adding a member id_t corresponds to inserting a new leaf into τ. The leaf's public signature and encryption keys are fetched from the AKS. Further, the new leaf becomes an *unmerged leaf* of all nodes on the path from it to the root. A leaf l being unmerged at a node v indicates that the l's owner doesn't know the secret key in v, so messages should be encrypted directly to l. When v's key is refreshed during an update, its set of unmerged leaves is cleared.

Key schedule. Apart from the ratchet tree, all group members store a number of shared symmetric keys, unique to the current epoch. These are: the *application secret* — the group key exported to the E2E application, the *membership key* used to authenticate sent messages and the *init key* — mixed in the next epoch's secrets for FS.

The secrets are derived when an epoch is created, i.e. after the implicit update following each modification. The update generates the last path secret s_n, which we now call the commit secret. Then, the following secrets are derived. First, the commit and the old epoch's init secrets are hashed together to obtain the *joiner secret*. Then, the *epoch secret* is obtained by hashing the joiner with the new epoch's *context*, which we explain next. (The context is not mixed directly with init and commit secrets, because the joiner secret is needed by new members; see below.) Finally, the new epoch's application, membership and init secrets are obtained by hashing the epoch secret with different labels.

The context includes all relevant information about the epoch, e.g. (the hash of) the ratchet tree (which includes the member set). The purpose of mixing it into the key schedule is ensuring that parties in different epochs derive independent epoch secrets.

Joining. When an id_t joins a group, the party inviting them encrypts to them two secrets under a key fetched from the AKS. First, this is id_t's path secret from the implicit update following the add. Second, this is the new joiner secret, from which id_t derives other epoch secrets. Importantly, the new member hashes the joiner with the context, which means that it agrees on the epoch's state with all current members transitioning to it.

5.2 Intuition for the SAIK Protocol

mmPKE. In ITK, a member performing an update generates a sequence of path secrets $s_1, \dots, s_n$ and encrypts each s_i to each public key from a set of recipient public keys S_i using regular encryption. In contrast, SAIK redraws its internal abstraction boundaries viewing the sequence of encryptions as a single call to mmPKE. This allows it to use the DDH-based mmPKE construction of [36]. Compared to ITK, this cuts the computational complexity of encrypting $\vec{m}$ and the ciphertext size in half (asymptotically as n grows).

Authentication. The goal of authentication is to make sure that a member accepts a message from id only if id knows 1) the signing key for the verification key stored in id's leaf in the current ratchet tree and 2) the current key schedule. In ITK, where every member gets the same message, this is achieved by simply signing it and MACing with the current membership key. In SAIK, to optimize bandwidth, the mailboxing service forwards to each receiver only the data it needs. E.g., it does not forward ciphretexts for other members. Therefore, we have to achieve authentication differently.

One trivial solution would be that the sender uploads multiple signatures, one for each receiver. However, this clearly does not scale. Can we do something better? A crucial observation is that the goal of saCGKA is to authenticate created epochs and *not message bitstrings*. That is, we want to guarantee that if Alice thinks that a message c transitions her to an epoch E created by Bob, then Bob indeed created E. It is not an attack if the adversary can make Alice accept a message that is not extracted with the honest procedure (e.g., it has reordered fields), as long as it transitions her to E.

Therefore, instead of signing the whole message, in SAIK we can sign and MAC only a single short tag that identifies the new epoch and is known to all members. In particular, this value is derived in the key schedule for the new epoch, alongside the other secrets, by hashing the epoch secret with an appropriate label. This way of efficient authentication is enabled by our new security notion.

Extraction procedure for the server. The task of the mailboxing server is to extract a personalized packet for a group member Alice from a packet C uploaded by another member Bob. In SAIK, C consists roughly of a single mmPKE ciphertext, a signature, the new public keys on the path from the sender to the root node and some metadata such as the sender's identity, the group modification being applied etc. The signature and metadata are simply forwarded to Alice. For the mmPKE ciphertext, the server runs the mmPKE Ext procedure with Alice's recipient index i and also sends all public keys up to the lowest common ancestor (lca) of Alice and Bob in the ratchet tree. See Fig. 1 for an illustration. Observe that i and the lca are determined by the current epoch's ratchet tree and the positions of Alice and Bob in it. Therefore, the server can obtain i and the lca in two ways: First, it can store all ratchet trees it needs (identified by the transcript hash leading to the epoch for which a tree is stored) and them itself. Second, it can ask Alice for i and the lca given that the sender is Bob. We note that the latter solution requires an additional round of interaction which may be problematic for some applications.

Comparison with techniques of [31]. The work [31] introduces a technique for efficient packet authentication which is quite similar to the technique used by SAIK. In particular, their CGKA uses a committing mPKE, cmPKE. A cmPKE differs from mPKE in that encryption outputs a tag T which is a cryptographic commitment to the plaintext and is delivered to each receiver. Since in [31] every recipient of a commit gets the same message, authenticating T is sufficient for CGKA authentication. We would like to highlight a couple of differences between that technique and ours: First, it is not clear how to use cmPKE in a tree-based CGKA, where a commit executes multiple instances of cmPKE, and hence we end up with multiple tags T, each delivered to a different subset of the group. Second, observe that using the hash of the encrypted message as T does not result in an IND-CCA secure cmPKE, since a hash allows to easily tell which of two messages is encrypted. Therefore, the cmPKE construction of [31] uses key-committing encryption to both hide and bind the message. SAIK 's solution is simpler and doesn't require additional properties.

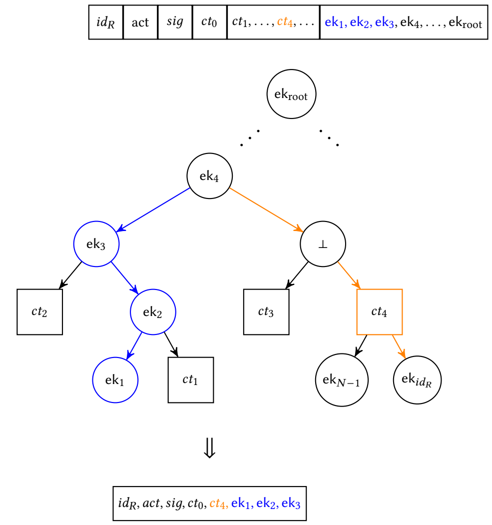


Figure 1: Server extraction algorithm. Lowest common ancestor (LCA) pof id_R and id_S is ek_4, so all blue public keys are included in id_R's packet. Since the sibling of ek_3 is empty, there corresponding path secret of ek_4 is encrypted to its resolution, resulting in the two ciphertext.

To summarize, cmPKE introduced by [31] is very useful for the CGKA type they consider and may well find more use-cases beyond CGKA. On the other hand, SAIK 's solution is simpler and fits all types of CGKA. Albeit, it is very CGKA-specific.

6 SECURITY OF SAIK

To define the security we prove for SAIK we fix the two safety predicates **confidential** and **authentic** used by $\mathcal{F}_{\text{CGKA}}$. We next give the intuition; see Fig. 17 in App. H for the pseudocode. We define two versions of the predicates: a stronger and a weaker one. For better exposition, the stronger version is not achieved by SAIK as presented in this work. But at the cost of added complexity SAIK can easily be extended to achieve it, as described Sec. 8.1.

We begin with the simpler stronger version. First, both predicates give no guarantees for epochs in detached trees until they are attached and so we ignore them in this section. Then, the definition is built around the notion of secrets which make up the protocol state. There are two types of secrets: group secrets, stored in the state of all parties, and individual secrets, stored in the states of some parties. Each corruption exposes a number of secrets and each epoch change replaces a number of secrets by (possibly) secure ones. The helper predicate `*grp-secs-secure`(E) decides if the group secrets in E are secure, i.e., not exposed, and the predicate `*ind-secs-secure`(E, id) decides if id's individual secrets in E are secure. Then **confidential**(E) equals to `*grp-secs-secure`(E), since the epoch key is itself a group secret. Further, **authentic**(E, id) is true if either `*grp-secs-secure`(E) or `*ind-secs-secure`(E, id) is true, because both group and id's secrets are necessary to impersonate id in E.

It remains to determine when group and individual secrets are exposed. For group secrets, `*grp-secs-secure`(E) is defined recursively. The base case states that the group secrets in first epoch (when the group was created) are secure if and only if no party is corrupted while in that epoch. Intuitively, we assume the group was created by an honest party using good randomness. Moreover, capturing perfect forward secrecy, corruptions in the descendant epochs do not affect the confidentiality of earlier group secrets.

The induction step states that the group secrets in a non-root epoch E are secure if no party is corrupted in E, the epoch is not created by an injected packet from the adversary and either the group secrets in E's parent E_p are secure or all individual secrets in E are secure. Intuitively, this formalizes the requirement that the adversary can learn the group secrets in only three ways: A) by corrupting a party currently in epoch E. B) by injecting the secrets (though most injections are disallowed by the authenticity predicate). C) by computing them the same way an honest *receiver* transitioning to E would. The latter requires knowing the group secrets of E_p and the individual secrets of at least one receiver. Note that the possible receivers are those parties that are group members in E and that are not E's creator (who transitions on sending). Note also that the fact that even knowing an epoch creator's individual secrets in E_p we can treat them as secure in E captures so called *post compromise security* (aka. *healing* or *backwards security*). Indeed, in SAIK, part of creating a new epoch requires refreshing all ones individual secrets.

Finally, individual secrets of id in E are exposed whenever there is some other epoch E' where id's secrets are the same as in E and where id was corrupted or its secrets were injected on its behalf. The secrets of id are the same in two epochs if no epoch between them replaces the secrets, i.e., is created by id, removes it or adds it.

Weaker guarantees. In the weaker version of the security predicates, individual secrets of id in E are not secure in an additional scenario, formalized by `*exposed-ind-secs-weak`. In this scenario, an id_s first honestly adds id and the adversary $\mathcal{A}$ injects a message adding id to some other epoch. Finally, id joins $\mathcal{A}$'s epoch and is corrupted before sending any message. We explain why SAIK is insecure in this case and how it can be modified to be secure in Sec. 8.1.

Security. For the mmPKE scheme we assume a security property called mmOW-RCCA, defined in App. G. The notion is strictly weaker than mmIND-CCA; in App. G we prove the implication.

We note that formally, the AKS is modeled as the functionality $\mathcal{F}_{\text{AKS}}$ defined in App. E. SAIK works in the $\mathcal{F}_{\text{AKS}}$-hybrids model, i.e., $\mathcal{F}_{\text{AKS}}$ is available in the real world and emulated by the simulator in the ideal world.

Theorem 6.1. *Let $\mathcal{F}_{\text{CGKA}}$ be the CGKA functionality with predicates* **confidential** *and* **authentic** *defined in Fig. 17. Let* SAIK *be instantiated with an mmPKE* mmPKE*, a signature scheme* Sig *and* MAC*, and with the* HKDF *functions modelled as a random oracle* Hash*. Let $\mathcal{A}$ be any environment. Denote the output of $\mathcal{A}$ from the real execution with* SAIK *and the hybrid functionality $\mathcal{F}_{\text{AKS}}$ from Fig. 13 as $\text{REAL}_{\mathsf{SAIK},\mathcal{F}_{\text{AKS}}}(\mathcal{A})$ and the output of $\mathcal{A}$ from the ideal execution with $\mathcal{F}_{\text{CGKA}}$ and a simulator $\mathcal{S}$ as $\text{IDEAL}_{\mathcal{F}_{\text{CGKA}},\mathcal{S}}(\mathcal{A})$. There exists a simulator $\mathcal{S}$ and adversaries $\mathcal{B}_1$ to $\mathcal{B}_4$ such that*

$$
\begin{aligned}
\Pr[\text{IDEAL}_{\mathcal{F}_{\text{CGKA}},\mathcal{S}}(\mathcal{A}) = 1] - \Pr\left[\text{REAL}_{\mathsf{SAIK},\mathcal{F}_{\text{AKS}}}(\mathcal{A}) = 1\right] \leq&\\
\mathrm{Adv}^{\mathsf{CR}}_{\mathsf{Hash}}(\mathcal{B}_1)&\\
+ q_e^2(q_e+1)\log(q_n)\cdot \mathrm{Adv}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE},q_e\log(q_n),q_n}(\mathcal{B}_2)&\\
+ 2q_e \cdot \mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig}}(\mathcal{B}_3)&\\
+ q_e \cdot \mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{B}_4) + 3q_h q_e^2(q_e+1)/2^{\kappa},&
\end{aligned}
$$

where q_e, q_n and q_h denote bounds on the number of epochs, the group size and the number of $\mathcal{A}$'s queries to the random oracle modelling the Hash*, respectively.*

7 EVALUATION

We compare the communication complexity (informally, the "bandwidth") of SAIK, ITK and the CmPKE protocol from [31]. For the sake of this comparison (and to simplify the description), one can think of CmPKE as a protocol similar to SAIK but where the ratchet tree is an N-ary tree of height 1, where N is the number of group members. This means that CmPKE only needs single-message multi-recipient PKE, mPKE (which is a special case of mmPKE). To make a fair comparison, we instantiate CmPKE with the same DH-based mPKE as SAIK instead of the less efficient but post-quantum secure mPKE given in [31].

Methodology. More precisely, we break down the communication complexity into the *sender bandwidth*, i.e., the size of a packet uploaded to the server, and the *receiver bandwidth*, i.e., the size of a (personalized) packet downloaded by a single receiver.

We note that for SAIK, the receiver bandwidth (i.e. the number of fields in the packet) differs depending on *the position of the receiver in the ratchet tree*, relative to the sender. (The reason is that only public keys on the path from the sender to the lowest common ancestor in the ratchet tree need to be downloaded.) Therefore, for all protocols we compare the average receiver bandwidth over all members of the group. (For ITK and CmPKE this makes no difference.)

Another difficulty in making a meaningful comparison is that for SAIK and ITK the bandwidth can vary quite significantly depending on *the execution history.* The reason is that add and remove operations may destroy the good properties of the ratchet tree, increasing the number of recipients to which some message must be encrypted. In the best case, there are only $\log(N)$ recipients. Roughly, this happens when the ratchet has no blank nodes or unmerged leaves; see Fig. 2. However, in the worst case there can be N recipients. This happens e.g. when all non-leaf nodes are blank; see Fig. 3. In general, the number of recipients can be anything in between; see Fig. 4.

Comparing the average over all execution histories would not be meaningful, since the probability of a given execution depends on the setting such as user and administrator behavior and general governing policies, which is outside the scope of this work. Indeed,

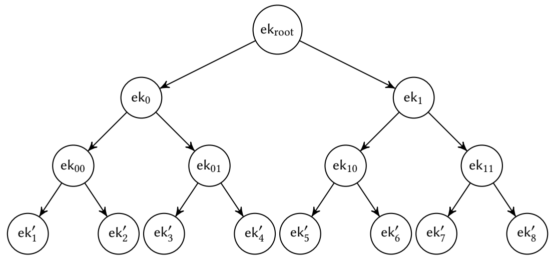


Figure 2: A ratchet tree for SAIK or ITK without blanks or unmerged leaves.

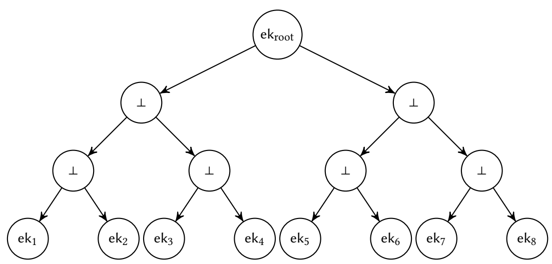


Figure 3: A ratchet tree for SAIK or ITK with all nodes blank.

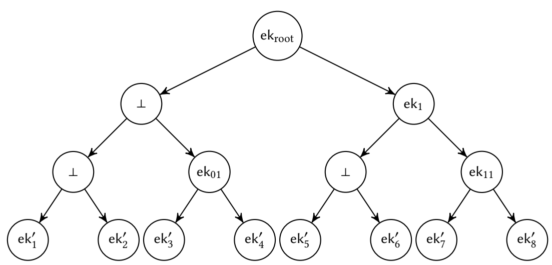


Figure 4: A tree with some blank and non-blank nodes. Here, sender bandwidth depends on the position in the tree. For example, a packet by the leftmost leaf would contain 5 ciphertexts, while the rightmost leaf would require 6.

it is an important topic of future research to better understand which kinds of policies governing when and which parties initiate CGKA operations lead to more bandwidth efficient executions for realistic deployments. Therefore, we compare the minimum and maximum bandwidth over all possible execution histories.

Results. We computed the bandwidth for all protocols using the formulas in Tables 1 and 2. Further, this is visualized in Fig. 5 for growing group sizes N.

We highlight some interesting features of the plots. For concreteness, we fix $N = 10K$ users. For senders, SAIK is always at least as good as the other protocols. It requires between 83% and only 55% of the bandwidth of ITK (due to the use of mmPKE). Compared to CmPKE, the sender bandwidth of SAIK is the same for the worst-case execution history and for the best-case execution, SAIK needs as little as 0.45% of the bandwidth, e.g. in a group of $10K$ members it can save as much as $\sim$ 780KB out of total 783KB.

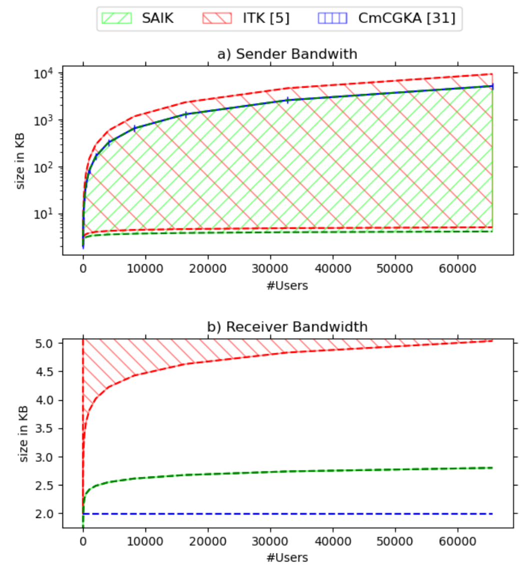


Figure 5: Bandwidth comparison of SAIK, ITK and CmPKE (instantiated with 256-bit security). Lower lines denote the best case execution history, while upper lines denote the worst case. All other possible cases are marked as the regions between the lines. Plot (a) shows the sender bandwidth on a *log scale* and plot (b) shows average individual receiver bandwidth on a *linear scale*. Note that in the first plot, the lines for worst-case SAIK and all-case CmPKE coincide.

For receivers, ITK is *overwhelmingly* worse than SAIK and CmPKE. For example, SAIK receivers (on average) need between 60% (best-case execution) to about .05% (worst-case execution) of ITK's. On the other hand, CmPKE is unsurprisingly the best for receivers. SAIK requires up to 137% of CmPKE's bandwidth, i.e. an increase of $\sim$ 0.75KB from $\sim$ 2KB for a group of $10K$ members.

Server computation. Lastly, we consider the server-side computation for SAIK and CmPKE. In CmPKE, the server only picks the i-th mPKE ciphertext for the i-th user and forwards all common data. For SAIK, we can consider two possibilities: Either the server keeps track of the shape of the ratchet tree (which it can do based on the header data sent in all packages) and computes the lowest common ancestor of sender and receiver in the tree, computes its resolution and then forwards the corresponding ciphertext and public keys. This takes at most logarithmic time in the size of the group (however, no expensive public-key operations are required). Alternatively, the user can compute its indices in the tree and send them to the server, reducing the server computation to effectively the same as in CmPKE at the cost of an additional round of communication.

8 EXTENSIONS

In this section we describe extensions of SAIK which we did not include for simplicity. The first extension allows to achieve slightly better security predicates at a relatively small cost. The second extension deals with primitives with imperfect correctness, such as mmPKE based on lattices.

		ITK	SAIK	CmPKE
Sender	best case	$\log(N) \cdot (\mathsf{Pk} + \mathsf{Ctx})$	$\log(N) \cdot \mathsf{Pk} + \mathsf{mCtx}(\log(N))$	$\mathsf{Pk} + \mathsf{mCtx}(N)$
	worst case	$\log(N) \cdot \mathsf{Pk} + N \cdot \mathsf{Ctx}$	$\log(N) \cdot \mathsf{ek} + \mathsf{mCtx}(N)$	$\mathsf{Pk} + \mathsf{mCtx}(N)$
(Average) receiver	best case	$\log(N) \cdot (\mathsf{Pk} + \mathsf{Ctx})$	$\log(N) \cdot \mathsf{Pk} + \mathsf{Ctx}$	$\mathsf{Pk} + \mathsf{Ctx}$
	worst case	$\log(N) \cdot \mathsf{Pk} + N \cdot \mathsf{Ctx}$	$\log(N) \cdot \mathsf{Pk} + \mathsf{Ctx}$	$\mathsf{Pk} + \mathsf{Ctx}$

Table 1: Sender and receiver bandwidth for a group of size N expressed as the number of ciphertexts and public keys included in the packet (apart from this, packets include only a constant-size header). Pk **denotes the size of a public key (the same for PKE and mmPKE).** $\mathsf{mCtx}(X)$ **denotes the size of an mmPKE multi-recipient ciphertext with overall number of receivers X. Note that for the DH-based construction X fully determines the size (i.e., it is not affected by who gets which message).** Ctx **denotes the size of a PKE ciphertext, equal to the size of an individual ciphertext in the DH-based construction.**

		ITK	SAIK	CmPKE
Sender	best case	$3\log(N)$	$2\log(N)$	N
	worst case	$2N$	N	N
(Average) receiver	best case	$3\log(N)$	$\log(N)$	3
	worst case	$2N$	$\log(N)$	3

Table 2: Sender and receiver bandwidth for a group of size N expressed as the (approximate) number of group elements.

8.1 Better Security Predicates

We sketch the reason why SAIK does not achieve the better security predicates and how it can be modified to achieve them.

Roughly, SAIK achieves the worse security predicates because of the following attack: Say id_s, the only corrupted party, creates a new epoch E adding a new member id. According to SAIK, in this case id_s fetches from the Authenticated Key Service, AKS, (a type of PKI setup) a public key ek for mmPKE and a verification key spk for Sig, both registered earlier by id. In epochs after E, parties use ek to encrypt messages to id (even before id actually joins) and spk to verify messages from id. Now the adversary $\mathcal{A}$ can create a fake epoch E' adding id with the same ek and spk. Then, id joins E' and is corrupted, leaking dk and ssk. This allows $\mathcal{A}$ to compute the group key in E and inject messages to parties in E. However, the expectation is that this is not possible, since no party is corrupted in E (and id_s healed). The better security predicates (formally, the predicates in Fig. 17 in **??**) achieve just this: security in an honest epoch E does not depend on whether some member joins a fake group in E'.

The following modification to SAIK achieves better security: We note that in SAIK, id registers in the AKS an additional public key ek' which is used to send secrets needed for joining. The corresponding dk' is deleted immediately after joining. In the modified SAIK, when id_s adds id, it generates for id new key pairs $(\mathsf{ek}_s, \mathsf{dk}_s)$ and $(\mathsf{spk}_s, \mathsf{ssk}_s)$. It sends dk_s and ssk_s to id, encrypted under ek'. Now messages to id are encrypted such that *both* dk and dk_s are needed to decrypt them. In particular, to encrypt m, a sender chooses a random r and encrypts r under ek and $m \oplus r$ under ek_s. Similarly, messages from id have two signatures, one verified under spk, and one under spk_s. As soon as id creates an epoch, it generates a new single mmPKE key pair and a single HRS key pair.

The attack is prevented, because even after corrupting id in E', $\mathcal{A}$ does not know dk' needed to decrypt dk_s and ssk_s. Therefore, confidentiality and authenticity in E is not affected.

8.2 Primitives with Imperfect Correctness

While the proofs of SAIK security assume primitives perfect correctness, they can be easily modified to work with imperfect correctness. This is achieved by adding one game hop where we abort in the new game if a correctness error occurs. This loses an additive term in the security bound that depends on the correctness parameter and the number of possible occurrences. Additionally, the usage of primitives with imperfect correctness generally yields imperfect correctness guarantees for the application as well (potentially with multiplicative correctness error when using multiple primitives). For completeness, we give definitions of imperfect correctness of the primitives used directly by SAIK in this section.

Definition 8.1. We call an mmPKE scheme δ-*correct*, if for all $n \in \mathbb{N}$, $(\mathsf{ek}_i, \mathsf{dk}_i) \in \mathsf{KG}$ for $i \in [n]$, $(m_1, \ldots, m_n) \in \mathcal{M}^n$ and $\forall j \in [n]$

$$\Pr\left[\begin{array}{c} c_j \leftarrow \mathsf{Ext}(j, C) \\ m_j \neq \mathsf{Dec}(\mathsf{dk}_j, c_j) \end{array} \middle| C \xleftarrow{\$} \mathsf{Enc}\left(\begin{array}{c} (\mathsf{ek}_1, \ldots, \mathsf{ek}_n), \\ (m_1, \ldots, m_n) \end{array}\right)\right] \leq \delta$$

REFERENCES

[1] Joël Alwen, Benedikt Auerbach, Mirza Ahad Baig, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. 2021. Grafting Key Trees: Efficient Key Management for Overlapping Groups. In *Theory of Cryptography — TCC 2021*. Full version: https://ia.cr/2021/1158.

[2] Joël Alwen, Bruno Blanchet, Eduard Hauck, Eike Kiltz, Benjamin Lipp, and Doreen Riepel. 2021. Analysing the HPKE Standard. In *Advances in Cryptology - EUROCRYPT 2021 - 40th Annual International Conference on the Theory and Applications of Cryptographic Techniques, Zagreb, Croatia, October 17-21, 2021, Proceedings, Part I*. 87–116. https://doi.org/10.1007/978-3-030-77870-5_4 Full Version: https://ia.cr/2020/1499.

[3] Joël Alwen, Margarita Capretto, Miguel Cueto, Chethan Kamath, Karen Klein, Ilia Markov, Guillermo Pascual-Perez, Krzysztof Pietrzak, Michael Walter, and Michelle Yeo. 2021. 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.

[4] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. 2019. The Double Ratchet: Security Notions, Proofs, and Modularization for the Signal Protocol. In *EUROCRYPT 2019, Part I (LNCS)*, Yuval Ishai and Vincent Rijmen (Eds.), Vol. 11476. Springer, Heidelberg, 129–158. https://doi.org/10.1007/978-3-030-17653-2_5

[5] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. 2020. Security Analysis and Improvements for the IETF MLS Standard for Group

Messaging. In *CRYPTO 2020, Part I (LNCS)*, Daniele Micciancio and Thomas Ristenpart (Eds.), Vol. 12170. Springer, Heidelberg, 248–277. https://doi.org/10.1007/978-3-030-56784-2_9

[6] Joël Alwen, Sandro Coretti, Yevgeniy Dodis, and Yiannis Tselekounis. 2021. Modular Design of Secure Group Messaging Protocols and the Security of MLS. In *ACM CCS 2021 (to appear)*. Full version: https://ia.cr/2021/1083.pdf.

[7] Joël Alwen, Sandro Coretti, Daniel Jost, and Marta Mularczyk. 2020. Continuous Group Key Agreement with Active Security. In *TCC 2020, Part II (LNCS)*, Rafael Pass and Krzysztof Pietrzak (Eds.), Vol. 12551. Springer, Heidelberg, 261–290. https://doi.org/10.1007/978-3-030-64378-2_10

[8] Joël Alwen, Daniel Jost, and Marta Mularczyk. 2020. On The Insider Security of MLS. Cryptology ePrint Archive, Report 2020/1327. (2020). https://eprint.iacr.org/2020/1327.

[9] Michael Backes, Markus Dürmuth, Dennis Hofheinz, and Ralf Küsters. 2006. Conditional Reactive Simulatability. In *ESORICS 2006 (LNCS)*, Dieter Gollmann, Jan Meier, and Andrei Sabelfeld (Eds.), Vol. 4189. Springer, Heidelberg, 424–443. https://doi.org/10.1007/11863908_26

[10] Richard Barnes. 06 August2018 13:01UTC. Subject: [MLS] Remove without double-join (in TreeKEM). MLS Mailing List. (06 August2018 13:01UTC). https://mailarchive.ietf.org/arch/msg/mls/Zzw2tqZC1FCbVZA9LKERsMIQXik.

[11] Richard Barnes. 22 August 2019 22:17UTC. Subject: [MLS] Proposal: Proposals (was: Laziness). MLS Mailing List. (22 August 2019 22:17UTC). https://mailarchive.ietf.org/arch/msg/mls/5dmrkULQeyvNu5k3MV_sXreybj0/.

[12] R. Barnes, B. Beurdouche, , J. Millican, E. Omara, K. Cohn-Gordon, and R. Robert. 2020. *The Messaging Layer Security (MLS) Protocol (draft-ietf-mls-protocol-latest)*. Technical Report. IETF. https://messaginglayersecurity.rocks/mls-protocol/draft-ietf-mls-protocol.html.

[13] Richard Barnes, Jon Millican, Emad Omara, Katriel Cohn-Gordon, and Raphael Robert. 2018. Message Layer Security (mls) WG. https://datatracker.ietf.org/wg/mls/about/. (2018).

[14] M. Bellare, A. Boldyreva, K. Kurosawa, and J. Staddon. 2007. Multirecipient Encryption Schemes: How to Save on Bandwidth and Computation Without Sacrificing Security. *IEEE Transactions on Information Theory* 53, 11 (2007), 3927–3943. https://doi.org/10.1109/TIT.2007.907471

[15] Mihir Bellare, Alexandra Boldyreva, and Jessica Staddon. 2003. Randomness Re-use in Multi-recipient Encryption Schemeas. In *PKC 2003 (LNCS)*, Yvo Desmedt (Ed.), Vol. 2567. Springer, Heidelberg, 85–99. https://doi.org/10.1007/3-540-36288-6_7

[16] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. 2018. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups. (May 2018). http://prosecco.inria.fr/personal/karthik/pubs/treekem.pdf Published at https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8.

[17] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. 2019. *Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS.* Research Report. Inria Paris. https://hal.inria.fr/hal-02425229

[18] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. 2020. On the Price of Concurrency in Group Ratcheting Protocols. Cryptology ePrint Archive, Report 2020/1171. (2020). https://ia.cr/2020/1171.

[19] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. 2021. Cryptographic Security of the MLS RFC, Draft 11. Cryptology ePrint Archive, Report 2021/137. (2021). https://ia.cr/2021/137.

[20] Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. 2016. Universal Composition with Responsive Environments. In *ASIACRYPT 2016, Part II (LNCS)*, Jung Hee Cheon and Tsuyoshi Takagi (Eds.), Vol. 10032. Springer, Heidelberg, 807–840. https://doi.org/10.1007/978-3-662-53890-6_27

[21] Ran Canetti. 2001. Universally Composable Security: A New Paradigm for Cryptographic Protocols. In *42nd FOCS*. IEEE Computer Society Press, 136–145. https://doi.org/10.1109/SFCS.2001.959888

[22] Ran Canetti, Hugo Krawczyk, and Jesper Buus Nielsen. 2003. Relaxing Chosen-Ciphertext Security. In *CRYPTO 2003 (LNCS)*, Dan Boneh (Ed.), Vol. 2729. Springer, Heidelberg, 565–582. https://doi.org/10.1007/978-3-540-45146-4_33

[23] Haitao Cheng, Xiangxue Li, Haifeng Qian, and Di Yan. 2018. CCA Secure Multi-recipient KEM from LPN. In *ICICS 18 (LNCS)*, David Naccache, Shouhuai Xu, Sihan Qing, Pierangela Samarati, Gregory Blanc, Rongxing Lu, Zonghua Zhang, and Ahmed Meddahi (Eds.), Vol. 11149. Springer, Heidelberg, 513–529. https://doi.org/10.1007/978-3-030-01950-1_30

[24] Cisco. 2021. Zero-Trust Security for Webex. (2021). https://www.cisco.com/c/en/us/solutions/collateral/collaboration/white-paper-c11-744553.pdf.

[25] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. 2018. On Ends-to-Ends Encryption: Asynchronous Group Messaging with Strong Security Guarantees. In *ACM CCS 2018*, David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang (Eds.). ACM Press, 1802–1819. https://doi.org/10.1145/3243734.3243747

[26] Ronald Cramer and Victor Shoup. 2002. Universal Hash Proofs and a Paradigm for Adaptive Chosen Ciphertext Secure Public-Key Encryption. In *EUROCRYPT 2002 (LNCS)*, Lars R. Knudsen (Ed.), Vol. 2332. Springer, Heidelberg, 45–64. https://doi.org/10.1007/3-540-46035-7_4

[27] Yevgeniy Dodis, Paul Grubbs, Thomas Ristenpart, and Joanne Woodage. 2018. Fast Message Franking: From Invisible Salamanders to Encryptment. In *CRYPTO 2018, Part I (LNCS)*, Hovav Shacham and Alexandra Boldyreva (Eds.), Vol. 10991. Springer, Heidelberg, 155–186. https://doi.org/10.1007/978-3-319-96884-1_6

[28] Nir Drucker and Shay Gueron. 2019. Continuous Key Agreement with Reduced Bandwidth. In *Cyber Security Cryptography and Machine Learning - Third International Symposium, CSCML 2019, Beer-Sheva, Israel, June 27-28, 2019, Proceedings (Lecture Notes in Computer Science)*, Shlomi Dolev, Danny Hendler, Sachin Lodha, and Moti Yung (Eds.), Vol. 11527. Springer, 33–46. https://doi.org/10.1007/978-3-030-20951-3_3

[29] Taher ElGamal. 1984. A Public Key Cryptosystem and a Signature Scheme Based on Discrete Logarithms. In *CRYPTO'84 (LNCS)*, G. R. Blakley and David Chaum (Eds.), Vol. 196. Springer, Heidelberg, 10–18.

[30] Keita Emura, Kaisei Kajita, Ryo Nojima, Kazuto Ogawa, and Go Ohtake. 2022. Membership Privacy for Asynchronous Group Messaging. Cryptology ePrint Archive, Report 2022/046. (2022). https://ia.cr/2022/046.

[31] Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. 2021. A Concrete Treatment of Efficient Continuous Group Key Agreement via Multi-Recipient PKEs. In *Proceedings of the 2021 ACM SIGSAC Conference on Computer and Communications Security*. 1441–1462.

[32] Daniel Jost, Ueli Maurer, and Marta Mularczyk. 2019. A Unified and Composable Take on Ratcheting. In *TCC 2019, Part II (LNCS)*, Dennis Hofheinz and Alon Rosen (Eds.), Vol. 11892. Springer, Heidelberg, 180–210. https://doi.org/10.1007/978-3-030-36033-7_7

[33] Shuichi Katsumata, Kris Kwiatkowski, Federico Pintore, and Thomas Prest. 2020. Scalable Ciphertext Compression Techniques for Post-quantum KEMs and Their Applications. In *ASIACRYPT 2020, Part I (LNCS)*, Shiho Moriai and Huaxiong Wang (Eds.), Vol. 12491. Springer, Heidelberg, 289–320. https://doi.org/10.1007/978-3-030-64837-4_10

[34] Kaoru Kurosawa. 2002. Multi-recipient Public-Key Encryption with Shortened Ciphertext. In *PKC 2002 (LNCS)*, David Naccache and Pascal Paillier (Eds.), Vol. 2274. Springer, Heidelberg, 48–63. https://doi.org/10.1007/3-540-45664-3_4

[35] A. Langley, M. Hamburg, and S. Turner. 2016. Elliptic curves for security. RFC 7748, RFC Editor. (2016).

[36] Alexandre Pinto, Bertram Poettering, and Jacob C. N. Schuldt. 2014. Multi-recipient encryption, revisited. In *ASIACCS 14*, Shiho Moriai, Trent Jaeger, and Kouichi Sakurai (Eds.). ACM Press, 229–238.

[37] Eric Rescorla. 03 May 2018 14:27UTC. Subject: [MLS] TreeKEM: An alternative to ART. MLS Mailing List. (03 May 2018 14:27UTC). https://mailarchive.ietf.org/arch/msg/mls/WRdXVr8iUwibaQu0tH6sDnqU1no.

[38] Nigel P. Smart. 2005. Efficient Key Encapsulation to Multiple Parties. In *SCN 04 (LNCS)*, Carlo Blundo and Stelvio Cimato (Eds.), Vol. 3352. Springer, Heidelberg, 208–219. https://doi.org/10.1007/978-3-540-30598-9_15

[39] Nick Sullivan. 29 January 2020 21:39UTC. Subject: [MLS] Virtual interim minutes. MLS Mailing List. (29 January 2020 21:39UTC). https://mailarchive.ietf.org/arch/msg/mls/ZZAz6tXj-jQ8nccf7SyIwSnhivQ/.

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

Supplementary Material

A ADDITIONAL PRELIMINARIES

A.1 Universal Composability

We formalize security in the universal composability (UC) framework [21]. We moreover use the modification of responsive environments introduced by Camenisch et al. [20] to avoid artifacts arising from seemingly local operations (such as sampling randomness or producing a ciphertext) to involve the adversary.

The UC framework requires a real-world execution of the protocol to be indistinguishable from an ideal world, to an an interactive environment. The real-world experiment consists of the group members executing the protocol (and interacting with the PKI setup). In the ideal world, on the other hand, the protocol gets replaced by dummy instances that just forward all inputs and outputs to an *ideal functionality* characterizing the appropriate guarantees.

The functionality interacts with a so-called simulator, that translates the real-world adversary's actions into corresponding ones in the ideal world. Since the ideal functionality is secure by definition, this implies that the real-world execution cannot exhibit any attacks either.

The Corruption Model. We use the — standard for CGKA/SGM but non-standard for UC — corruption model of continuous state leakage (transient passive corruptions) [7].[7] In a nutshell, this corruption model allows the adversary to repeatedly corrupt parties by sending corruption messages of the form (Expose, id), which causes the party id to send its current state to the adversary (once).

Restricted Environments. In order to avoid the so-called commitment problem, caused by adaptive corruptions in simulation-based frameworks, we restrict the environment not to corrupt parties at certain times. (This roughly corresponds to ruling out "trivial attacks" in game-based definitions. In simulation-based frameworks, such attacks are no longer trivial, but security against them requires strong cryptographic tools and is not achieved by most protocols.) To this end, we use the technique used in [7] (based on prior work by Backes et al. [9] and Jost et al. [32]) and consider a weakened variant of UC security that only quantifies over a restricted set of so-called admissible environments that do not exhibit the commitment problem. Whether an environment is admissible or not is defined as part of the ideal functionality $\mathcal{F}$: The functionality can specify certain boolean conditions, and an environment is then called admissible (for $\mathcal{F}$), if it has negligible probability of violating any such condition when interacting with $\mathcal{F}$.

A.2 Assumptions

The security of our mmPKE construction, same as that of [36], is based on a variant of the Computational Diffie-Hellman(CDH) assumption called the *Double-Sided Strong Diffie-Hellman Assumption* (or just *Static Diffie-Hellman Assumption* in [36]). We recall it in Definition A.1. Intuitively, it states that CDH is hard given access to a DDH-oracle for both CDH inputs.

[7] Passive corruptions together with full network control allow to emulate active corruptions.

Game mmIND-RCCA

$\underline{\mathbf{Exp}^{\text{mmIND-RCCA}}_{\text{mmPKE},N,b}(\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2))}$

- **for** $i \in [N]$ **do** $(\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{KG}()$
- $\mathsf{Corr} \leftarrow \varnothing$
- $(\vec{\mathsf{ek}}^*, \vec{m}^*_0, \vec{m}^*_1, st) \leftarrow \mathcal{A}_1^{\mathbf{Dec}_1, \mathbf{Cor}}((\mathsf{ek}_i)_{i \in [N]})$
- **req** $|\vec{m}^*_0| = |\vec{m}^*_1| = |\vec{\mathsf{ek}}^*|$
- $c^* \xleftarrow{\$} \mathsf{Enc}(\vec{\mathsf{ek}}^*, \vec{m}^*_b)$
- $b' \leftarrow \mathcal{A}_2^{\mathbf{Dec}_2, \mathbf{Cor}}(c^*, st)$
- **req** $\mathsf{leak}(\vec{m}_0) = \mathsf{leak}(\vec{m}_1)$
- **req** $\forall j : \vec{\mathsf{ek}}^*[j] \in \{\mathsf{ek}_i : i \in [N]\} \setminus \mathsf{Corr} \lor m^*_0[j] = m^*_1[j]$
- **return** b'

$\underline{\textbf{Oracle } \mathbf{Dec}_1(i, c)}$

- **req** $i \in [N]$
- **return** $\mathsf{Dec}(\vec{\mathsf{dk}}[i], c)$

$\underline{\textbf{Oracle } \mathbf{Cor}(i)}$

- **req** $i \in [N]$
- $\mathsf{Corr} \mathrel{+}\!\!\leftarrow i$
- **return** dk_i

$\underline{\textbf{Oracle } \mathbf{Dec}_2(i, c)}$

- **req** $i \in [N]$
- $m \leftarrow \mathsf{Dec}(\vec{\mathsf{dk}}[i], c)$
- **if** $\exists j : \vec{\mathsf{ek}}^*[j] = \mathsf{ek}_i$
  $\land\ m \in \{\vec{m}^*_0[j], \vec{m}^*_1[j]\}$ **then**
  - **return** *'test'*
- **else return** m

Figure 6: mmIND-RCCA **security game for** mmPKE **with leakage function** $\mathsf{leak}(\vec{m}) = (\mathtt{len}(\vec{m}[1]), \ldots, \mathtt{len}(\vec{m}[n]))$**.**

Definition A.1 (Double-Sided Strong Diffie-Hellman Assumption). Let $\mathcal{G} = (\mathbb{G}, p, g)$ be a cyclic group of prime order p with generator g. We define the advantage of an algorithm $\mathcal{A}$ in solving the *Double-Sided Strong Diffie-Hellman problem(*DSSDH*)* with respect to $\mathcal{G}$ as

$$\mathrm{Adv}^{\mathrm{DSSDH}}_{\mathcal{G}}(\mathcal{A}) = \left[Z = g^{xy} \;\middle|\; \begin{array}{c} x, y \xleftarrow{\$} \mathbb{Z}_p^2 \\ Z \xleftarrow{\$} \mathcal{A}^{\mathbf{O}}(\mathbb{G}, p, g, g^x, g^y), \end{array} \right]$$

with $\mathbf{O} = \{\mathbf{O}_x(\cdot,\cdot), \mathbf{O}_y(\cdot,\cdot)\}$, where $\mathbf{O}_x, \mathbf{O}_y$ are oracles which on input U, V output 1, iff $U^x = V$ or $U^y = V$ respectively. The probability is taken over the random coins of the group generator, the choice of x and y and the adversary's random coins.

A.3 Multi-Recipient Multi-Message PKE(mmPKE) Definitions

The notion of mmIND-RCCA security for mmPKE is described by the experiment in Fig. 6.

We define the security of an mmPKE in a left-right style in the following Theorem A.2.

*Definition A.2 (*mmIND-RCCA*).* Let $N \in \mathbb{N}$. For a scheme mmPKE, we define the advantage of an adversary $\mathcal{A}$ against *Indistinguishability Against Replayable Chosen Ciphertext Attacks (*mmIND-RCCA*)* security of mmPKE as

$$\begin{aligned} \mathrm{Adv}^{\text{mmIND-RCCA}}_{\text{mmPKE},N}(\mathcal{A}) = \Pr\left[\mathrm{Exp}^{\text{mmIND-RCCA}}_{\text{mmPKE},N,0}(\mathcal{A}) = 1\right] \\ - \Pr\left[\mathrm{Exp}^{\text{mmIND-RCCA}}_{\text{mmPKE},N,1}(\mathcal{A}) = 1\right], \end{aligned}$$

where $\mathrm{Exp}^{\text{mmIND-RCCA}}_{\text{mmPKE},N,b}$ is described in Fig. 6.

A.4 Data Encapsulation Meachanism(DEM)

A DEM is the symmetric equivalent of a PKE scheme. We recall it in Theorem A.3.

Definition A.3 (DEM). A data encapsulation mechanism (DEM) DEM is described by a (efficiently samplable) keyspace $\mathcal{K}$ and the two algorithms $\mathsf{D}, \mathsf{D}^{-1}$:

$\mathsf{D}(k, m) \xrightarrow{\$} c$: The encryption algorithm takes a key $k \in \mathcal{K}$ and a message m. It returns a ciphertext c.

$\mathsf{D}^{-1}(k, c) \xrightarrow{\$} m' \vee \perp$: The decryption algorithm takes a key $k \in \mathcal{K}$ and a ciphertext c and outputs either a decrypted message or $\perp$.

A DEM DEM is δ-correct, if for all messages m and all keys $k \in \mathcal{K}$

$$\Pr[\mathsf{D}^{-1}(k, \mathsf{D}(k, m)) = m] \geq \delta$$

Analogue to mmPKE, we consider IND-RCCA security for DEMs. It is described in Theorem A.4.

Definition A.4. The advantage of an adversary $\mathcal{A}$ against the IND-RCCA security of a DEM DEM is defined as

$$\mathrm{Adv}^{\mathsf{IND\text{-}RCCA}}_{\mathsf{DEM}}(\mathcal{A}) = \Pr[\mathrm{Exp}^{\mathsf{IND\text{-}RCCA}}_{\mathsf{DEM},0}(\mathcal{A}) = 1] - \Pr[\mathrm{Exp}^{\mathsf{IND\text{-}RCCA}}_{\mathsf{DEM},1}(\mathcal{A}) = 1],$$

where $\mathrm{Exp}^{\mathsf{IND\text{-}RCCA}}_{\mathsf{DEM},b}(\mathcal{A})$ is defined in Fig. 7.

Game IND-RCCA **for** DEM

$\underline{\mathbf{Exp}^{\mathsf{IND\text{-}RCCA}}_{\mathsf{DEM},b}(\mathcal{A})}$

$k \xleftarrow{\$} \mathcal{K}$
$(St, m_0^*, m_1^*) \xleftarrow{\$} \mathcal{A}^{\mathrm{Dec,Enc}}$
$c^* \xleftarrow{\$} \mathsf{D}(k, m_b)$
return $\mathcal{A}^{\mathrm{Dec,Enc}}(St, c^*)$

$\underline{\textbf{Oracle Dec}(c)}$

$m' \xleftarrow{\$} \mathsf{D}^{-1}(k, c)$
if $m' \in \{m_0^*, m_1^*\}$ **then**
  return test
else
  return m'

$\underline{\textbf{Oracle Enc}(m)}$

return $\mathsf{D}(k, m)$

Figure 7: IND-RCCA **security for DEMs.**

A.5 Message Authentication Codes (MAC)

Message authentication codes are defined in Theorem A.5.

Definition A.5. A message authentication code MAC = (MAC.tag, MAC.vrf) consist of a keyspace $\mathcal{K}$ and the following two algorithms:

$\mathsf{MAC.tag}(k, m) \xrightarrow{\$} \mathsf{tag}$: The tagging algorithm takes a key k and a message m and outputs a tag t.

$\mathsf{MAC.vrf}(k, m, \mathsf{tag}) \xrightarrow{\$} \{0, 1\}$: The verification algorithm takes a key k, a message m and a tag tag and outputs 0 or 1.

A mac MAC is correct, if for all $k \in \mathcal{K}$ and messages m

$$\Pr[\mathsf{MAC.vrf}(k, m, \mathsf{MAC.tag}(k, m)) = 1] = 1$$

The security notion for MACs we consider is *Unforgeability against chosen message attacks*(EUF-CMA).

Definition A.6. A mac MAC is EUF-CMA secure, if for all PPT adversaries $\mathcal{A}$ the advantage

$$\mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{A}) = \Pr\left[\begin{array}{c} m \notin Q \wedge \\ \mathsf{MAC.vrf}(k, m^*, \mathsf{tag}^*) = 1 \end{array} \middle| \begin{array}{c} k \xleftarrow{\$} \mathcal{K} \\ (m^*, t^*) \xleftarrow{\$} \mathcal{A}^{\mathrm{Tag,Ver}} \end{array}\right]$$

is negligible, where the Tag oracle computes a tag under key k on a given message m and adds it to Q and *Ver* takes a message and a tag and outputs the result of the MAC.vrf algorithm on the two inputs with k.

B NOMINAL GROUPS

We recall the definition and parameters for nominal groups from [2].

Definition B.1 (Nominal Group). A nominal group $\mathcal{N} = (\mathcal{G}, g, p, \mathcal{E}_H, \mathcal{E}_U, \exp)$ consists of a finite set of elements $\mathcal{G}$, a base element $g \in \mathcal{G}$, a prime p, a finite set of "good" exponents $\mathcal{E}_H \subset \mathbb{Z}$, a set of exponents $\mathcal{E}_U \subset \mathbb{Z} \setminus p\mathbb{Z}$ and an efficiently computable exponentiation function $\exp : \mathcal{G} \times \mathbb{Z} \to \mathcal{G}$. We write X^y as shorthand for $\exp(X, y)$ and call elements of $\mathcal{G}$ "group elements". $\mathcal{N}$ has to fulfil the following properties:

(1) $\mathcal{G}$ is efficiently recognizable.
(2) $(X^y)^z = X^{yz}$ for all $X \in \mathcal{G}, y, z \in \mathbb{Z}$
(3) the function ϕ defined by $\phi(x) = g^x$ is a bijection from $\mathcal{E}_U$ to $\{g^x | x \in [p-1]\}$.

A nominal group $\mathcal{N}$ is called *rerandomisable*, when additionally

(4) $g^{x+py} = g^x$ for all $x, y \in \mathbb{Z}$
(5) for all $y \in \mathcal{E}_U$, the function ϕ_y defined by $\phi_y(x) = g^{xy}$ is a bijection from $\mathcal{E}_U$ to $\{g^x | x \in [p-1]\}$.

Property 3 (and 5) ensure that discrete logarithms are unique in $\mathcal{N}$ in $\mathcal{E}_U$. Additionally, we define the two statistical parameters

$$\Delta_{\mathcal{N}} := \Delta[G_H, G_U],$$

with G_H is the uniform distribution over $\mathcal{E}_H$ and G_U is the uniform distribution over $\mathcal{E}_U$ and

$$P_{\mathcal{N}} = \max_{Y \in \mathcal{G}} \Pr_{x \xleftarrow{\$} \mathcal{E}_H}[Y = g^x].$$

Any cyclic group, such as NIST curves, can be seen as a rerandomisable nominal group with the special properties that $\Delta_{\mathcal{N}} = 0$ and $P_{\mathcal{N}} = p - 1$. Other popular examples of rerandomisable nominal groups are Curve25519 and Curve448. Table 3 lists the parameters for these nominal groups.

For a more detailed explanation of these values, see [2]. Nominal groups and prime-order groups behave indistinguishably except when group elements are sampled with exponents outside of $\mathcal{E}_H$ or a collision occurs which wouldn't have been a collision in a prime-order group. Since these two events are statistical in nature and occur with low probability, this only adds a negligible additive security loss compared to Theorem 3.2.

The DSSDH assumption is almost identical over nominal groups except for the choice of exponents.

Definition B.2 (Double-Sided Strong Diffie-Hellman Assumption). Let $\mathcal{N} = (\mathcal{G}, g, p, \mathcal{E}_H, \mathcal{E}_U, \exp)$ be a nominal group. We define the

Name	**P-256**	**P-384**	**P-512**	**Curve25519**	**Curve448**
Security Level	128	192	256	128	224
$P_{\mathcal{N}}$	2^{-255}	2^{-383}	2^{-520}	2^{-250}	2^{-444}
$\Delta_{\mathcal{N}}$	0	0	0	2^{-125}	2^{-220}
Size in bits	256	384	512	256	512

Table 3: Statistical parameters of NIST curves and nominal group curves.

advantage of an algorithm $\mathcal{A}$ in solving the *Double-Sided Strong Diffie-Hellman problem(*DSSDH*)* with respect to $\mathcal{N}$ as

$$\mathrm{Adv}_{\mathcal{N}}^{\mathsf{DSSDH}}(\mathcal{A}) = \left[Z = g^{xy} \middle| \begin{array}{c} x, y \xleftarrow{\$} \mathcal{E}_U^2 \\ Z \xleftarrow{\$} \mathcal{A}^{\mathbf{O}}(\mathcal{N}, p, g, g^x, g^y), \end{array} \right]$$

with $\mathbf{O} = \{\mathbf{O}_x(\cdot,\cdot), \mathbf{O}_y(\cdot,\cdot)\}$, where $\mathbf{O}_x, \mathbf{O}_y$ are oracles which on input U, V output 1, iff $U^x = V$ or $U^y = V$ respectively. The probability is taken over the random coins of the group generator, the choice of x and y and the adversaries random coins.

Remark 2. Since x, y are sampled from $\mathcal{E}_U$, the second property of nominal groups guarantees that the oracles $\mathbf{O}_x$ and $\mathbf{O}_y$ are well-defined.

Theorem B.3. *Let* $\mathcal{N} = (\mathcal{G}, g, p, \mathcal{E}_H, \mathcal{E}_U, \exp)$ *be a nominal group. If the* DSSDH *assumption holds relative to* $\mathcal{N}$ *and* DEM *is an* IND-RCCA *secure DEM, then* mmPKE *from Fig. 8 is* mmIND-RCCA *secure with adaptive corruptions in the random oracle model and leakage function* leak *revealing the length of each plaintext. Specifically, there are adversaries* $\mathcal{B}_1, \mathcal{B}_2$ *against* DSSDH *and* IND-RCCA *of* DEM *respectively, s.t. for all adversaries* $\mathcal{A}$ *against the* mmIND-RCCA

$$\begin{aligned} &\mathrm{Adv}_{\mathsf{mmPKE}}^{\mathsf{mmIND\text{-}RCCA}}(\mathcal{A}) \leq \\ &\quad 2e^2 q_C \cdot n \cdot (\mathrm{Adv}_{\mathcal{N}}^{\mathsf{DSSDH}}(\mathcal{B}_1) + \frac{q_{D_1}}{p} + \frac{q_H}{p}) \\ &\quad + n \cdot \mathrm{Adv}_{\mathsf{DEM}}^{\mathsf{IND\text{-}RCCA}}(\mathcal{B}_2) \\ &\quad + 2(n+1)^2 \cdot \Delta_{\mathcal{N}}, + O(q_D, q_H) \cdot P_{\mathcal{N}}, \end{aligned}$$

where the runtime of $\mathcal{B}_1$ *and* $\mathcal{B}_2$ *is roughly the same as* $\mathcal{A}$ *and* q_{D_1}, q_H *and* q_C *denote the number of queries to the decryption oracle* D *in phase 1, the random oracle* H *and the corruption oracle* Cor *respectively.*

Proof. Mainly, the proof of Theorem 3.2 still applies. That is because all operations performed by the adversary are well-defined over nominal groups. The main difference occurs when rerandomising the keys in each hybrid. Here, not every exponent yields a valid group element, i.e. a valid key. Formally, we would add an additional hybrid for each chosen key, sampling its exponent from $\mathcal{E}_U$ instead of $\mathcal{E}_H$, which adds an additive term in $\Delta_{\mathcal{N}}$ to the advantage function. It is imperative that $\mathcal{N}$ is rerandomisable as otherwise embedding the (randomised) challenge would be problematic.

Secondly, whenever group elements are submitted to one of the oracles, there is a (tiny) probability of collisions of group elements. As it is comparable to the chances of guessing discrete logarithms in prime order groups, which is mostly ignored in proofs, we omit a complete analysis as it wouldn't contribute any meaningful insights.

Algorithm DH-mmPKE[$\mathbb{G}, g, p$, DEM, Hash]

<u>KG</u>
$x \xleftarrow{\$} \mathbb{Z}_p$
return $(\mathsf{dk} = g^x, \mathsf{ek} = x)$

<u>Dec($\mathsf{dk}_i, c = (c_0, c_i)$)</u>
$k = \mathsf{Hash}(c_0^{\mathsf{dk}_i}, \mathsf{ek}_i, i)$
return $\mathsf{DEM.D}^{-1}(k, c_i)$

<u>Enc($\vec{\mathsf{ek}}, \vec{m}$)</u>
$r \xleftarrow{\$} \mathbb{Z}_p$
for $i \in [|\vec{m}|]$ **do**
  $k \leftarrow \mathsf{Hash}(\mathsf{ek}_i^r, \mathsf{ek}_i, i)$
  $c_i = \mathsf{DEM.D}(k, m_i)$
return $(c_0 = g^r, c_1, \ldots, c_n)$

<u>Ext($i, C = (c_0, \ldots, c_n)$)</u>
return (c_0, c_i)

Figure 8: The mmPKE scheme based on Diffie-Hellman from [36]. The scheme requires a group $\mathbb{G}$ of prime order p generated by g, a data encapsulation mechanism DEM **and a hash function** Hash.

In conclusion, after sampling all keys from $\mathcal{E}_U$ and accounting for possible collisions in the gap oracles, the proof for nominal groups works as shown in Theorem 3.2. □

C SECURITY PROOF FOR THE MMPKE CONSTRUCTION

Theorem 3.2. *Let* $\mathbb{G}$ *be a group of prime order* p *with generator* g*, let* DEM *be a data encapsulation mechanism and let* mmPKE = DH-mmPKE[$\mathbb{G}, g, p$, DEM, H]*. For any adversary* $\mathcal{A}$ *and any* $N \in \mathbb{N}$*, there exist adversaries* $\mathcal{B}_1$ *and* $\mathcal{B}_2$ *with runtime roughly the same as* $\mathcal{A}$*'s s.t.*

$$\begin{aligned} \mathrm{Adv}_{\mathsf{mmPKE},N}^{\mathsf{mmIND\text{-}RCCA}}(\mathcal{A}) &\leq \mathrm{Adv}_{\mathsf{DEM}}^{\mathsf{IND\text{-}RCCA}}(\mathcal{B}_2) \\ &\quad + 2n \cdot (e^2 q_c \mathrm{Adv}_{(\mathbb{G},g,p)}^{\mathsf{DSSDH}}(\mathcal{B}_1) + \frac{q_{d_1}}{p} + \frac{q_h}{p}), \end{aligned}$$

where H *is modeled as a random oracle,* e *is the Euler number,* n *is the length of the challenge vector, and* q_{d_1}*,* q_c *and* q_h *are the number of queries to the decrypt and corrupt oracles and the random oracle, respectively.*

Proof. We define n hybrids G_0 through G_n, where G_0 is identical to $\mathrm{Exp}_{\mathsf{mmPKE},N,0}^{\mathsf{mmIND\text{-}RCCA}}$, G_n is identical to $\mathrm{Exp}_{\mathsf{mmPKE},N,1}^{\mathsf{mmIND\text{-}RCCA}}$ and in G_i, the first i challenge ciphertexts contain encryptions of $\vec{m}_1$ and the others from $\vec{m}_0$.

Additionally, for $i \in [n]$, we define the four hybrids $G_{i,0}$ to $G_{i,3}$. $G_{i,0}$ and $G_{i,3}$ are identical to G_i and G_{i+1} respectively. In $G_{i,1}$, we set the i-th DEM key to a random key and in $G_{i,2}$ we swap the plaintext in the i-th challenge ciphertext from $\vec{m}_0[i]$ to $\vec{m}_1[i]$.

We will split the proofs into the following lemmas.

Game $G_{i,1}$

Game $G_{i,1}$

$\textbf{for } i \in [N] \textbf{ do } (\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{KG}$
$\mathsf{Corr} \leftarrow \emptyset, \mathsf{HL} \leftarrow \emptyset$
$(\vec{\mathsf{ek}}^*, \vec{m}_0^*, \vec{m}_1^*, st) \leftarrow \mathcal{A}_1^{\mathrm{Dec_1,Cor,H}}(g, \mathsf{ek}_1, \ldots, \mathsf{ek}_N)$
Parse $\vec{\mathsf{ek}}^*$ as $\hat{\mathsf{ek}}_{i_1}, \ldots, \hat{\mathsf{ek}}_{i_l}, \mathsf{ek}^*_{l+1}, \ldots, \mathsf{ek}^*_n$ for $l \in [n]$
s.t. $\forall j \in [l] : \vec{m}_0[i_j] \neq \vec{m}_1[i_j] \wedge \hat{\mathsf{ek}}_{i_j} \in \vec{\mathsf{ek}}$
$\textbf{req } |\vec{m}_0^*| = |\vec{m}_1^*| = |\vec{\mathsf{ek}}^*| = n$
$r \xleftarrow{\$} \mathbb{Z}_p, c_0^* \leftarrow g^r$
$\textbf{for } 1 \leq j \leq n \textbf{ do }]\ K_j \leftarrow H(\mathsf{ek}_j^r, \mathsf{ek}_j, j)$
$K^* \xleftarrow{\$} \mathcal{K}$
$K_i \leftarrow K^*$
$\textbf{for } 1 \leq j \leq i \textbf{ do } c_j^* = \mathsf{D}(K_j, \vec{m}_1^*[j])$
$\textbf{for } i \leq j \leq n \textbf{ do } c_j^* = \mathsf{D}(K_j, \vec{m}_0^*[j])$
$c^* \leftarrow (c_0^*, \ldots, c_n^*)$
$b \leftarrow \mathcal{A}^{\mathrm{Dec_2,Cor,H}}(c^*, st)$
$\textbf{req } \forall j \in [n] : \vec{\mathsf{ek}}^*[j] \notin \{\mathsf{ek} : i \in [N] \setminus \mathsf{Corr}\} \implies$
$m_0^*[j] = m_1^*[j]$
$\textbf{return } b$

Oracle $\mathbf{Dec}_1(i, (c_0, c_i))$

$\textbf{req } i \in [N]$
$K \leftarrow H(c_0^{\mathsf{dk}_i}, \mathsf{ek}_i, i)$
$\textbf{return } \mathsf{D}^{-1}(K, c_i)$

Oracle $\mathbf{Dec}_2(j, c = (c_0, (c_k, k)))$

$\textbf{req } j \in [N]$
$K \leftarrow H(c_0^{\vec{\mathsf{dk}}[j]}, \vec{\mathsf{ek}}[j], k)$
$\textbf{if } c_0 = c_0^* \wedge \vec{\mathsf{ek}}^*[k] = \vec{\mathsf{ek}}[j] \wedge i \leq l \textbf{ then}$
$\quad K \leftarrow K_j^*$
$m \leftarrow \mathsf{D}^{-1}(K, c_k)$
$\textbf{if } \exists k : \vec{\mathsf{ek}}^*[j] = \mathsf{ek}_k \wedge m \in \{\vec{m}_0^*[j], \vec{m}_1^*[j]\}$
then
$\quad$ **return** `test`
else
$\quad$ **return** m

Oracle $\mathbf{Cor}(i)$

$\textbf{req } i \in [N]$
$\mathsf{Corr} \mathrel{+}\leftarrow i$
$\textbf{return } \mathsf{dk}_i$

Oracle $\mathbf{H}(Z, W, i)$

$\textbf{if } \mathsf{HL}[Z, W, i] = \perp \textbf{ then}$
$\quad \mathsf{HL}[Z, W, i] \xleftarrow{\$} \mathcal{K}$
$\textbf{return } \mathsf{HL}[Z, W, i]$

Figure 9: Description of the hybrid $G_{i,1}$

Lemma C.1. *Let $n \in \mathbb{N}$. For all $1 \leq i \leq n$, there exists an adversary $\mathcal{B}_1$ against the* DSSDH *assumption s.t. for all adversaries $\mathcal{A}$*

$$|\Pr[G_{i,0}(\mathcal{A}) \Rightarrow 1] - \Pr[G_{i,1}(\mathcal{A}) = 1]| \leq e^2 q_C \cdot \mathrm{Adv}_{\mathcal{G}}^{\mathsf{DSSDH}}(\mathcal{B}_1) + \frac{q_{D_1}}{p} + \frac{q_H}{p},$$

where q_H, q_C and q_{D_1} denote the number of hash queries, corruption queries and decryption queries in phase 1 respectively made by $\mathcal{A}$.

Remark 3. Since the changes from $G_{i,2}$ to $G_{i,3}$ are the same as from $G_{i,0}$ to $G_{i,1}$, Theorem C.1 applies there as well.

Lemma C.2. *Let $n \in \mathbb{N}$. Then for all $1 \leq i \leq n$, there exists an adversary $\mathcal{B}_2$ against the* IND-RCCA *security of* DEM *s.t. for all adversaries $\mathcal{A}$*

$$|\Pr[G_{i,1}^{\mathcal{A}} \Rightarrow 1] - \Pr[G_{i,2}^{\mathcal{A}} \Rightarrow 1]| \leq \mathrm{Adv}_{\mathsf{DEM}}^{\mathsf{IND\text{-}RCCA}}(\mathcal{B}_2)$$

Combining the two lemmas and the remark yields Theorem C.3. The theorem follows by a standard hybrid argument over G_i.

Corollary C.3. *Let $n \in \mathbb{N}$. Then for all $1 \leq i \leq n$, there exist adversaries $\mathcal{B}_1, \mathcal{B}_2$ against the* DSSDH *assumption and the* IND-RCCA *security of* DEM *respectively s.t. for all adversaries $\mathcal{A}$*

$$|\Pr\left[G_i^{\mathcal{A}} \Rightarrow 1\right] - \Pr\left[G_{i+1}^{\mathcal{A}} \Rightarrow 1\right]| \leq \mathrm{Adv}_{\mathsf{DEM}}^{\mathsf{IND\text{-}RCCA}}(\mathcal{B}_2) + 2(e^2 q_C \cdot \mathrm{Adv}_{\mathcal{G}}^{\mathsf{DSSDH}}(\mathcal{B}_1) + \tfrac{q_{D_1}}{p} + \tfrac{q_H}{p})$$

with q_{D_1}, q_C and q_H from Theorem C.1.

So all that is left is proving the lemmas. We will start by proving Theorem C.1. Consider the formal definition of $G_{i,1}$ in Fig. 9.

Next, we describe adversary $\mathcal{B}_1$ against the DSSDH assumption in Fig. 10 First, we argue that $\mathcal{B}_1$ simulates the game $G_{i,1}$ perfectly, unless one of the events Bad_1 or Bad_2 occurs. We will bound the probabilities of these events happening. The games differ only in the secret key of the i-th message, therefore $G_{i,0}$ and $G_{i,1}$ are identical to $\mathcal{A}$, unless it queries the hash oracle on (Z, U, i) as in line 1 of H. If the corresponding public key was a key in which the challenge was embedded, i.e. Bad_3 is false, $\mathcal{B}_1$ breaks the DSSDH assumption.

Bad_1 and Bad_2 prevent that $\mathcal{A}$ already knows the challenge randomness before its challenge query. If it doesn't know this value, all answers of $\mathcal{B}_1$ to both oracles are distributed as in the real games $G_{i,0}$ and $G_{i,1}$. Specifically, the oracles are kept consistent and the hash oracle is programmed such that the keys chosen for the adversaries challenge are included at the right points.

Bad_3 occurs if $\mathcal{A}$ tries to corrupt a public key for which $\mathcal{B}_1$ doesn't know the corresponding secret key or $\mathcal{A}$ chooses a key without the challenge embedded for the i-th message. If the first part doesn't happen, the corruption oracle is simulated perfectly. The adversary doesn't notice the second part in this case, but if it occurs, $\mathcal{B}_1$ isn't successful, so it is still a bad case for the simulation.

Now we bound the probability of these events occurring. Since Y is completely hidden from $\mathcal{A}$, it can only find it by guessing. Therefore, for an adversary $\mathcal{A}$ making at most q_H (resp. q_{D_1}) hash (resp. decryption) queries,

$$\Pr[\mathsf{Bad}_1 = \mathrm{True}] \leq \frac{q_H}{p}$$
$$\Pr[\mathsf{Bad}_2 = \mathrm{True}] \leq \frac{q_{D_1}}{p}$$

For Bad_3, consider the probability with which $b[j] = 1$. This is independent for each public key ek_j, so the probability that Bad_3 does *not* occur is the case that for q_C public keys $\mathsf{ek}_{i_1}, \ldots, \mathsf{ek}_{i_{q_C}}$

Algorithm Adversary $\mathcal{B}_1$ on DSSDH

Adversary $\mathcal{B}_1(\mathbb{G}, p, g, U, V)$

Corr $\leftarrow \emptyset$, HL $\leftarrow \emptyset$, DL $\leftarrow \emptyset$
for $j \in [N]$ **do**
  Pick $b[j] \leftarrow \{0,1\}$ with $\Pr[b[j] = 1] = \frac{1}{q_C}$
  $\alpha_j \xleftarrow{\$} \mathbb{Z}_p \setminus \{0\}$
  if $b[i] = 1$ **then** $\mathsf{ek}_j \leftarrow V^{\alpha_j}$ // Embed the challenge
  else $\mathsf{ek}_j \leftarrow g^{\alpha_j}$ // Allow corruption
Phase $\leftarrow 1$
$(\vec{m}_0^*, \vec{m}_1^*, \vec{\mathsf{ek}}^*, st) \xleftarrow{\$} \mathcal{A}^{\mathsf{H}, \mathsf{D}_1, \mathsf{Cor}}(\mathsf{ek}_1, \ldots, \mathsf{ek}_N)$
req $|\vec{m}_0| = |\vec{m}_1| = |\vec{\mathsf{ek}}^*| = n$
Parse $\vec{\mathsf{ek}}^*$ as $\hat{\mathsf{ek}}_{i_1}, \ldots, \hat{\mathsf{ek}}_{i_l}, \mathsf{ek}^*_{l+1}, \ldots, \mathsf{ek}^*_n$ for $l \in [n]$
s.t. $\forall j \in [l] : \vec{m}_0[i_j] \neq \vec{m}_1[i_j] \wedge \hat{\mathsf{ek}}_{i_j} \in \vec{\mathsf{ek}}$
if $b[i_i] = 0$ **then**
  $\mathsf{Bad}_3 \leftarrow True$
  abort
$c_0^* \leftarrow U$
for $j \in [n]$ **do**
  $K_j \xleftarrow{\$} \mathcal{K}$
  $c_j^* \leftarrow \mathsf{D}(K_j, \vec{m}_b[j])$
  if $\exists k \in [N] : \mathsf{ek}_k = \vec{\mathsf{ek}}^*[j] \wedge k < i \wedge b[k] = 1$ **then**
    $\mathsf{DL}[j, U, (c_j, j)] = K_j$
  if $j > i \wedge b[j] = 0$ **then**
    $\mathsf{HL}[U^{\alpha_j}, \mathsf{ek}^*[j], j] \leftarrow K_j$
Phase $\leftarrow 2$
$b' \xleftarrow{\$} \mathcal{A}^{\mathsf{H}, \mathsf{D}_2, \mathsf{Cor}}(c_0^*, \ldots, c_n^*, st)$
return $\perp$

Oracle $\mathsf{Cor}(j)$

Corr $+\!\!\leftarrow j$
if $b[j] = 1$ **then**
  $\mathsf{Bad}_3 \leftarrow True$
  abort
return α_j

Oracle $H(Z, W, j)$

if $\exists k \in [N] : W = \mathsf{ek}_k \wedge \mathsf{ek}_k = \mathsf{ek}^*[i] \wedge b[k] = 1 \wedge \mathbf{O}_v(U^{\alpha_k}, Z) = 1$
then return $Z^{\frac{1}{\alpha_k}}$
if Phase $= 1 \wedge \mathbf{O}_u(W, Z) = 1$ **then**
  $\mathsf{Bad}_2 \leftarrow 1$
  abort
if Phase $= 2 \wedge j \in [n] \wedge W = \mathsf{ek}^*[j] \wedge \mathbf{O}_u(W, Z) = 1 \wedge j > i$ **then**
  return K_j
if $\exists j \in [N], c \in \mathbb{G}, t \in \mathcal{K} : W = \mathsf{ek}_j \wedge \mathsf{DL}[i, (c, (*, j))] = t \wedge$
$\mathbf{O}_v(c^{\alpha_j}, Z) = 1$ **then**
  return t
if $\mathsf{HL}[Z, W, j] = \perp$ **then**
  $\mathsf{HL}[Z, W, j] \xleftarrow{\$} \mathcal{K}$
return $\mathsf{HL}[Z, W, j]$

Oracle $D_{\mathsf{Phase}}(i, (c_0, (c, j)))$

req $i \in [N]$
if Phase $= 1 \wedge c_0 = U$ **then**
  $\mathsf{Bad}_1 \leftarrow 1$
  abort
if $\exists Z \in \mathbb{G}, t \in \mathcal{K} : \mathsf{HL}[Z, \mathsf{ek}_j, j] = t \wedge (b[j] = 0 \implies Z = \mathsf{ek}_j^{\alpha_j} \wedge$
$b[j] = 1 \implies \mathbf{O}_v(c^{\alpha_j}, Z) = 1)$ **then**
  $m \leftarrow \mathsf{D}^{-1}(t, c)$
  if $\exists k \in [N] : \vec{\mathsf{ek}}^*[j] = \mathsf{ek}_k \wedge m \in \{\vec{m}_0^*[j], \vec{m}_1^*[j]\}$ **then**
    return test
  else
    return m
if $\mathsf{DL}[i, (c_0(c, j))] = \perp$ **then**
  $\mathsf{DL}[i, (c_0, (c, j))] \xleftarrow{\$} \mathcal{K}$
$m \leftarrow \mathsf{D}^{-1}(\mathsf{DL}[i, (c_0, j)], c)$
if $\exists k \in [N] : \vec{\mathsf{ek}}^*[j] = \mathsf{ek}_k \wedge m \in \{\vec{m}_0^*[j], \vec{m}_1^*[j]\}$ **then**
  return test
else
  return m

Figure 10: Description of adversary $\mathcal{B}_1$ from Theorem 3.2.

$b[i_j] = 0$ and for one public key the bit is 1, so

$$\Pr[\mathsf{Bad}_3 = \mathsf{False}] = (1 - \frac{1}{q_C})^{q_C} \cdot \frac{1}{q_C} \overset{(1)}{\leq} \frac{1}{e^2 q_C}$$

For (1), we use that $\ln(1 + x) \geq \frac{x}{x+1}$ for all $x \geq -1$ and rewrite $(1 - \frac{1}{q_C})^{q_C} = e^{\ln((1-\frac{1}{q_C})^{q_C})} = e^{q_C \cdot \ln(1-\frac{1}{q_C})} \geq e^{-1/(1-\frac{1}{q_C})} \geq e^{-2}$ for $q_C > 1$. Combining the probabilities yields the lemma.

The proof of Theorem C.2 is a straight forward application of the IND-RCCA security of the DEM. Since the key at position i is random, an IND-RCCA adversary can simulate encryptions for this position with its encryption oracle and embeds its own challenge at the i-th challenge ciphertext for the adversary $\mathcal{A}$. If $\mathcal{A}$ can distinguish between $G_{i,1}$ and $G_{i,2}$ then $\mathcal{B}_2$ distinguishes its challenges as well. □

D DETAILS OF THE (SA)CGKA SECURITY MODEL

In this section, we formally define $\mathcal{F}_{\text{CGKA}}$. The code of $\mathcal{F}_{\text{CGKA}}$ is in Figs. 11 and 12.

Notation. We use the keyword **assert** *cond* to restrict the simulator's actions. Formally, if the condition *cond* is false, then the functionality permanently halts, making the real and ideal worlds easily distinguishable. Further, we use **only allowed if** *cond* to restrict the environment. That is, our statements quantify only over environments who, when interacting with $\mathcal{F}_{\text{CGKA}}$ and any simulator, never make *cond* false. [8] Finally, we write *receive from the simulator* to denote that the functionality sends a dummy value to it, waits until it sends a value back and asserts via **assert** that the received value is of the correct format.

[8]A relaxed restriction would require that $\mathcal{A}$ makes *cond* false with a small probability ϵ. In our case $\mathcal{A}$ knows if it violates *cond*, so fixing $\epsilon = 0$ is without loss of generality.

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

Parameters: **confidential**(epid), **authentic**(epid, id), $\text{id}_{\text{creator}}$

```
Initialization // Executed on first input.
  CurEp[∗], HG[∗] ← ⊥
  epCtr ← 0
  E ← *new-ep; E.sndr ← id_creator; E.mem ← {id_creator}
  HG[0] ← E
  CurEp[id_creator] ← 0

Input (Send, act), act ∈ {'up', 'add'-id_t, 'rem'-id_t} from id
  // Send inputs to sim. and allow it to reject them.
  Send (Send, id, act) to the sim. and receive ack.
  req ack
  // Compute the new epoch E created by the action.
  E ← *new-ep; E.par ← CurEp[id]; E.sndr ← id
  E.act ← act; E.mem ← *mem(CurEp[id], act)
  epCtr++; HG[epCtr] ← E
  // Enforce security after possible changes to HG.
  assert *HG-is-consistent ∧ *auth-is-preserved
  // Immediately transition id into the created epoch.
  CurEp[id] ← epCtr
  // Output the idealized message chosen by adv.
  Receive from the simulator C.
  return C

Input GetKey from id
  Send (Key, id) to the simulator and receive I.
  epid ← CurEp[id]
  req epid ≠ ⊥
  if HG[epid].key = ⊥ then
    // Set the key according to confidential.
    if confidential(epid) then
      HG[epid].key ←$ {0,1}^κ
      HG[epid].chall ← true
    else
      HG[epid].key ← I
  return HG[epid].key

Corruption (Expose, id)
  if CurEp[id] ≠ ⊥ then // Record exposure.
    HG[CurEp[id]].exp +← id
  // Disallow adaptive corruptions to avoid commitment problem.
  only allowed if ∄epid : HG[epid].chall ∧ ¬confidential(epid)

Input (Receive, c) from id
  // Send inputs to sim. and allow it to reject them.
  Send (Receive, id, c) to the simulator and receive ack.
  req ack
  // Ask sim. to interpret the packet.
  Receive from the simulator (sndr', act').
  if act' = 'rem'-id then
    // Check that sndr' removed id or auth. not guaranteed.
    honestRem ← ∃epid : (HG[epid].par = CurEp[id]
                   ∧ HG[epid].sndr = sndr' ∧ HG[epid].act = 'rem'-id)
    assert honestRem ∨ ¬authentic(HG[CurEp[id]], sndr')
    CurEp[id] ← ⊥
    return (sndr', act')
  // Ask sim, to identify the epoch epid where id transitions. If epid = ⊥, a
  new injected epoch is created.
  Receive from the simulator epid.
  if epid = ⊥ then
    E ← *new-ep
    E.sndr ← sndr'; E.act ← act'; E.inj ← true
    if CurEp[id] ≠ ⊥ then
      // If id is a member, compute E.par and E.mem using its epoch.
      E.par ← CurEp[id]
      E.mem ← *mem(CurEp[id], act')
    else
      // If id joined, E is a detached root with arbitrary member set.
      Receive E.mem from the simulator; set E.par ← ⊥.
    epCtr++; HG[epCtr] ← V
    epid ← epCtr
  assert HG[epid] ≠ ⊥
  // If a current group member transitions to a detached root, attach it.
  if CurEp[id] ≠ ⊥ ∧ HG[epid].par = ⊥ then HG[epid].par ← CurEp[id]
  assert CurEp[id] = ⊥ ∨ HG[epid].par = CurEp[id]
  assert CurEp[id] ≠ ⊥ ∨ HG[epid].act = 'add'-id
  // Enforce security after possible changes to HG.
  assert *HG-is-consistent ∧ *auth-is-preserved
  // Transition id and compute its output.
  CurEp[id] ← epid
  if HG[epid].act = 'add'-id then return (HG[epid].sndr, HG[epid].mem)
  else return (HG[epid].sndr, HG[epid].act)
```

Figure 11: The ideal CGKA functionality.

State. $\mathcal{F}_{\text{CGKA}}$ maintains a history graph represented as an array HG, where HG[epid] denotes the epoch identified by an integer epid. We use the standard object-oriented notation for epochs. In particular, each epoch E has a number of attributes listed in Table 4 (E.inj, E.exp and E.chall are related to corruptions). Apart from HG, $\mathcal{F}_{\text{CGKA}}$ stores an array CurEp, where CurEp[id] denotes the current epoch of the party id.

Inputs from parties. The first two inputs, Send and Receive, are handled quite similarly. First, all inputted values are sent to the simulator (there are no private inputs). Second, the simulator sends a flag *ack* which decides if sending/receiving succeeds (or fails with output ⊥). Third, $\mathcal{F}_{\text{CGKA}}$ updates the history graph and enforces that this does not destroy authenticity and consistency by checking that `*auth-is-preserved` and `*HG-is-consistent` are true. Finally, $\mathcal{F}_{\text{CGKA}}$ transitions the sender/receiver to the new epoch (or removes its pointer in case it is removed) and computes the output using the new graph.

One aspect that needs more explanation is updating the graph when a party id receives c. In this case, the simulator interprets c for

Functionality $\mathcal{F}_{\text{CGKA}}$: Additional Methods

Helper `*new-ep`

return new epoch with sndr = ⊥, par = ⊥, act = ⊥, mem = ∅, inj = `false`, key = ⊥, exp = ∅, chall = `false`.

Helper `*mem`(epid, act)

$G \leftarrow$ HG[epid].mem
if act = *'add'*-id_t **then**
  G +← id_t
else if act = *'rem'*-id_t **then**
  G -← id_t
if act ≠ *'up'* ∧ G = HG[epid].mem **then**
  return ⊥
return G

Helper `*HG-is-consistent`

// True if HG is a forest and membership is consistent.
return true iff
  a) ∀id s.t. CurEp[id] ≠ ⊥ : id ∈ HG[CurEp[id]].mem
  b) HG has no cycles
  c) ∀epid ∈ [epCtr] : HG[epid].mem ≠ ⊥
  d) ∀epid ∈ [epCtr] s.t. HG[epid].par ≠ ⊥ :
    HG[epid].mem = `*mem`(HG[epid].par, HG[epid].act)

Helper `*auth-is-preserved`

// True if there is no authentic epoch created by injected packet. Observe that the root epid = 0 cannot be injected by definition.
return ∄epid : 1 ≤ epid ≤ epCtr ∧ HG[epid].inj
  ∧ **authentic**(HG[epid].par, HG[epid].sndr)

Figure 12: The ideal CGKA functionality: additional methods.

E.par	The integer identifier of the parent epoch.
E.sndr	The party who created the epoch by performing a group operation.
E.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.mem	The set of group members.
E.key	The shared group key.
E.inj	A boolean flag indicating if the epoch is injected.
E.exp	The set of group members exposed (i.e., corrupted) in this epoch.
E.chall	A flag indicating if a random group key has been outputted.

Table 4: Attributes on an epoch in $\mathcal{F}_{\text{CGKA}}$.

.

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

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

Initialization
  SK[·, ·] ← ⊥

Input `GetSK`(PK) **from** id
  SK ← SK[id, PK]
  SK[id, PK] ← ⊥
  return SK

Input (`GetPK`, id′) **from** id
  (PK, SK) ← `*AKS-kgen`()
  SK[id′, PK] ← SK
  Send (id′, PK) to adv.
  return PK

Figure 13: The Authenticated Key service Functionality.

$\mathcal{F}_{\text{CGKA}}$ (which abstracts away ciphertexts) by providing the sender sndr′ and the action act′. If act′ removes id, then the only possible authenticity check is that either sndr′ removed id in its current epoch or the epoch is not authentic for sndr′. If id is not removed, the simulator identifies the epoch epid into which id transitions or joins. The epoch can be ⊥, in which case $\mathcal{F}_{\text{CGKA}}$ creates a new epoch E with the infected flag inj set. If id is a current group member, then E is a child of its current epoch. Otherwise, if id joins, then E is a detached root. Afterwards, $\mathcal{F}_{\text{CGKA}}$ checks if epid identifies a detached root into which a current group member id transitions. If this is the case, the root is attached as a child of id's current epoch. For instance, this implies that any other party transitioning to epid must do so from id's current epoch and the epoch semantic must be consistent between it, id and the party who joined into epid.

The last input to $\mathcal{F}_{\text{CGKA}}$ is GetKey, which simply outputs the group key from the party's current epoch. The key is set to a random or arbitrary value the first time it is retrieved by some party.

Corruptions. When a party is corrupted, $\mathcal{F}_{\text{CGKA}}$ simply adds it to the exposed set exp of its current epoch. The set is later used by the security predicates. Then $\mathcal{F}_{\text{CGKA}}$ disallows corruptions in case extending the exp set switched **confidential** of some epoch E with E.chall set from true to false.

E THE AUTHENTICATED KEY SERVICE FUNCTIONALITY (AKS)

The AKS is modeled as the functionality $\mathcal{F}_{\text{AKS}}$ in Fig. 13. Formally, SAIK works in the $\mathcal{F}_{\text{AKS}}$-hybrids model, i.e., $\mathcal{F}_{\text{AKS}}$ is available in the real world and emulated by the simulator in the ideal world.

$\mathcal{F}_{\text{AKS}}$ works as follows. When a party id wants to fetch a key package of another party id′, $\mathcal{F}_{\text{AKS}}$ generates a new key package for id′ using SAIK's key-package generation algorithm (formally, the algorithm is a parameter of $\mathcal{F}_{\text{AKS}}$). It sends (the public part of) the package to id and to the adversary. (Note that since $\mathcal{F}_{\text{AKS}}$ exists in the real world, the adversary should be thought of as the UC environment.) The secrets for the key package can be fetched by id_t later, when it decides to join the group. Once fetched, secrets are deleted, which means that $\mathcal{F}_{\text{AKS}}$ cannot be used as secure storage.

F DETAILS OF SAIK

In this section we give the details of the SAIK protocol. The pseudocode can be found in Figs. 14 and 15.

F.1 Ratchet Trees

A ratchet tree is a left-balanced q-ary tree, formally defined in App. H.5. This generalizes ITK' binary trees. Using $q \neq 2$ can be beneficial in certain situations. A ratchet tree, as well as its nodes, have a number of labels listed in **??**. We also define a number of helper methods in **??**. Importantly, the *direct path* of a leaf u consists of (the ordered list of) all nodes on the path from u to the root, without u. The *resolution* of a node v is the minimal set of descendant non-blank nodes that covers the whole sub-tree rooted at v.

F.2 SAIK State and Algorithms

The state of SAIK consists of a number of variables, outlined in **??**. The protocol will ensure that states of any two parties in the same epoch differ at most in labels of nodes of $\gamma.\tau$ that describe secret keys and the label γ.leaf. This means that they agree on the secrets γ.appSec and γ.initSec, as well as on the public context, computed by the helper method in **??**.

SAIK's algorithms are defined in Figs. 14 and 15. Apart from initialization, there are three main algorithms (the rest of the code are subroutines) exposed to a user (or a higher-level application). They are identified by keywords `Send`, `Receive` and `Key`, respectively. First, `Send` is used to create a new epoch. When the user inputs `Send` followed by the intended group modification (update, add or remove), the protocol applies the modification and returns a message, which the user can upload to the mailboxing service. Second, `Receive` is used to process messages downloaded from the service. Third, with `Key` user gets the current group key.

The formal syntax of saCGKA protocols is defined as part of our security definition in App. D. In particular, an saCGKA protocol must expose the same interface as the ideal CGKA functionality.

F.3 Extraction Procedure for the Server

Finally, we describe a procedure `*extract`$(C, \mathrm{id}) \to c$ used by the mailboxing service to take an uploaded message C and compute the message c delivered to user id. Formally, this procedure is not part of our syntax or security definitions, since for simplicity our model does not consider correctness (see Sec. 4.6) and an untrusted service can anyway deliver arbitrary messages.

Recall that C contains the executed group operation act and the sender id_s, a multi-recipient ciphertext $Ctxt$ and a vector of updated public keys updEKs. Roughly, `*extract` only needs to compute id's individual mmPKE ciphertext mmPKE.Ext$(Ctxt, i)$ and the prefix of the first j elements of updEKs. This requires that it knows the indices i and j for id. We notice that they can be easily computed using the public part of the ratchet tree, act and id_s. Therefore, the indices can be obtained in two ways. First, the service can send act and id_s to id, who replies with i and j. This requires interaction, but both id and the service are online at the time. Second, the service can store the current ratchet trees and compute i and j itself. The disadvantage of this is that it requires keeping a large state — in case members are out of sync (e.g. a user is 10 epochs behind), the service needs to store one tree for each epoch which has an active member in it. Once i and j are known, `*extract` works as follows.

If act = *'up'*, set $ctxt$ = mmPKE.Ext$(Ctxt, i)$ and updEKs$'$ = (updEKs$[1], \ldots,$ updEKs$[j]$). Output c = (id_s, act, $ctxt$, updEKs$'$, sig), where sig is a field of C. If act = *'rem'*-id, output c = (id_s, *'removed'*, sig_t, tag_t) where sig_t and tag_t are taken from C. Finally, if act = *'add'*-id, C contains welcomeData, which in turn contains a ratchet tree. Based on this, compute id's index i in $Ctxt$, the number n of recipients of $Ctxt$, and then $ctxt_1$ = mmPKE.Ext$(Ctxt, i)$ and $ctxt_2$ = mmPKE.Ext$(Ctxt, n{+}1)$. Output c = (id_s, act, $ctxt_1$, $ctxt_2$, welcomeData).

G ONE-WAYNESS SECURITY OF MMPKE

In this section, we define One-Wayness under Relaxed Chosen Ciphertext Attacks security of mmPKE, mmOW-RCCA. Moreover, we prove that mmOW-RCCA security is implied by mmIND-RCCA security for schemes with large message spaces.

Motivation. We note that one-wayness security for mmPKE is less straightforward to define than for standard PKE schemes. Roughly, for standard PKE, one-wayness requires that given an encryption of a random message chosen by the challenger, no adversary can find the encrypted message. For mmPKE, the input to encryption is not a single message but a vector of messages. Moreover, even if the adversary corrupts recipients of some messages in the vector, it still should not be able to find the remaining messages. Therefore, it is now less clear how the challenge message vector should be chosen. The definition presented in this section is precisely what is needed for the security proof of ITK. We do not claim that it is the "right" notion, as it may not be suited to other applications.

The game. The mmOW-RCCA game is defined in Fig. 16, the challenge ciphertext is computed as follows: The adversary sends a public-key vector, as well as a message vector $\vec{m}$ and a set of indices S within this vector. The challenger then inserts the same random message m^* into all positions in $\vec{m}$ indicated by S (the previous values of $\vec{m}$ at these positions are ignored). It encrypts the result and sends the ciphertext to the adversary, whose goal is to find m^*.

Remarks. First, we note that the mmOW-RCCA game has no notion of leakage. Instead, the leakage is implicit in how the vector encrypted by the challenger is chosen — the "leakage" is everything the adversary knows about that vector, such as whether two slots contain the same message or not.

Second, the game allows the adversary to verify if some message m' is the correct solution m^*. This can be done by sending m' to the decrypt oracle and checking if it returns *'test'*. This additional ability makes the notion stronger (i.e., more difficult to achieve). We show that mmIND-RCCA security is sufficient to achieve it.

*Definition G.1 (*mmOW-RCCA*).* For an mmPKE with message space $\mathcal{M}$, the advantage of an adversary $\mathcal{A}$ against *One-Wayness under Replayable Chosen Ciphertext Attacks (*mmOW-RCCA*)* security of mmPKE is defined as

$$\mathrm{Adv}^{\text{mmOW-RCCA}}_{\text{mmPKE},N}(\mathcal{A}) = \Pr\left[\mathrm{Exp}^{\text{mmOW-RCCA}}_{\text{mmPKE},N}(\mathcal{A}) \Rightarrow 1\right],$$

where $\mathrm{Exp}^{\text{mmOW-RCCA}}_{\text{mmPKE},N}(\mathcal{A})$ is defined in Fig. 16.

Relation to mmIND-CCA. We prove that mmOW-RCCA security is implied by mmIND-RCCA for schemes with large message spaces.

SAIK: Algorithms

Initialization

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

Input $(\mathsf{Send}, \mathsf{act})$, $\mathsf{act} \in \{$ '*up*', '*rem*'-id_t, '*add*'-$\mathsf{id}_t\}$ **from** id

req $\gamma \neq \bot$
// In case of add, fetch id_t's keys from AKS (AKS runs `*AKS-kgen`).
if act = '*add*'-id_t **then**
  $(\mathsf{ek}_t, \mathsf{vk}_t, \mathsf{ek}'_t) \leftarrow$ query $(\mathsf{GetPk}, \mathsf{id}_t)$ to $\mathcal{F}_{\mathrm{KS}}$
  act $\leftarrow$ '*add*'-id_t-$(\mathsf{ek}_t, \mathsf{vk}_t, \mathsf{ek}'_t)$
// Create the state and secrets for the new epoch.
try $(\gamma', \mathsf{pathSecs}, \mathsf{joinerSec}) \leftarrow$ `*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**
  $\mathit{Ctxt} \leftarrow$ `*encrypt`$(\gamma', \mathsf{pathSecs}, \bot, \bot, \bot)$
else if act = '*add*'-id_t-$(\mathsf{ek}_t, \mathsf{vk}_t, \mathsf{ek}'_t)$ **then**
  $\mathit{Ctxt} \leftarrow$ `*encrypt`$(\gamma', \mathsf{pathSecs}, \mathsf{id}_t, \mathsf{ek}'_t, \mathsf{joinerSec})$
ssk $\leftarrow \gamma.\tau.\mathsf{leafof}(\mathsf{id}).\mathsf{ssk}$
sig $\leftarrow \mathsf{Sig.sign}(\mathsf{ssk}, \underline{\gamma'.\mathsf{confTag}})$
if act = '*rem*'-id_t **then**
  // Authenticate removal message for id_t
  $\mathsf{sig}_t \leftarrow \mathsf{Sig.sign}(\mathsf{ssk}, (\mathsf{id}, \text{'}\mathit{rem}\text{'-}\mathsf{id}_t))$
  $\mathsf{tag}_t \leftarrow \mathsf{MAC.tag}(\gamma.\mathsf{membKey}, (\mathsf{id}, \text{'}\mathit{rem}\text{'-}\mathsf{id}_t, \underline{\gamma.\mathsf{confTag}}))$
  return $(\mathsf{id}, \mathsf{act}, \underline{\mathit{Ctxt}}, \mathsf{updEKs}, \mathsf{sig}, \mathsf{sig}_t, \mathsf{tag}_t)$
$\gamma \leftarrow \gamma'$
if act = '*add*'-id_t-$(\mathsf{ek}_t, \mathsf{vk}_t, \mathsf{ek}'_t)$ **then**
  // Send additional data for id_t.
  welcomeData $\leftarrow (\gamma.\mathsf{grpId}, \gamma.\tau.\mathsf{public}(), \mathsf{ek}'_t)$
  return $(\mathsf{id}, \mathsf{act}, \underline{\mathit{Ctxt}}, \mathsf{updEKs}, \mathsf{sig}, \mathsf{welcomeData})$
return $(\mathsf{id}, \mathsf{act}, \underline{\mathit{Ctxt}}, \mathsf{updEKs}, \mathsf{sig})$

Input Key **from** id

req $\gamma \neq \bot$
$k \leftarrow \gamma.\mathsf{appSec}$
$\gamma.\mathsf{appSec} \leftarrow \bot$
return k

Input $(\mathsf{Receive}, (\mathsf{id}_s, \text{'}\mathit{removed}\text{'}, \mathsf{sig}_t, \mathsf{tag}_t))$ **from** id
// Receiver is removed.

vk $\leftarrow \gamma.\tau.\mathsf{leafof}(\mathsf{id}_s).\mathsf{vk}$
req $\mathsf{Sig.vrf}(\mathsf{vk}, (\mathsf{id}_s, \text{'}\mathit{rem}\text{'-}\mathsf{id}), \mathsf{sig}_t)$
req $\mathsf{MAC.vrf}(\gamma.\mathsf{membKey}, (\mathsf{id}_s, \text{'}\mathit{rem}\text{'-}\mathsf{id}, \underline{\gamma.\mathsf{confTag}}), \mathsf{tag}_t)$
$\gamma \leftarrow \bot$
return $(\mathsf{id}_s, \text{'}\mathit{rem}\text{'-}\mathsf{id})$

Input $(\mathsf{Receive}, (\mathsf{id}_s, \mathsf{act}, \underline{\mathit{ctxt}}, \underline{\mathsf{updEKs}'}, \mathsf{sig}))$ **from** id
// Receiver is a member.

try $\gamma' \leftarrow$ `*apply-act`$(\gamma.\mathsf{clone}(), \mathsf{id}_s, \mathsf{act})$
try $(\gamma, \mathsf{confTag}) \leftarrow$ `*transition`$(\gamma', \underline{\mathit{ctxt}}, \underline{\mathsf{updEKs}'}, \mathsf{id}_s, \mathsf{act})$
vk $\leftarrow \gamma.\tau.\mathsf{leafof}(\mathsf{id}_s).\mathsf{vk}$
req $\mathsf{Sig.vrf}(\mathsf{vk}, \underline{\mathsf{confTag}}, \mathsf{sig})$
if act = '*add*'-id_t-$(\mathsf{ek}_t, \mathsf{vk}_t)$ **then return** $(\mathsf{id}_s, \text{'}\mathit{add}\text{'-}\mathsf{id}_t)$
else return $(\mathsf{id}_s, \mathsf{act})$

Input $(\mathsf{Receive}, (\mathsf{id}_s, \mathsf{act}, \mathit{ctxt}_1, \mathit{ctxt}_2, \mathsf{welcomeData}))$ **from** id
// Receiver joins.

req $\gamma = \bot$
parse $(\mathsf{grpId}, \tau, \mathsf{ek}') \leftarrow \mathsf{welcomeData}$
$\gamma \leftarrow$ `*new-state`
$(\gamma.\mathsf{grpId}, \gamma.\tau, \gamma.\mathsf{lastAct}) \leftarrow (\mathsf{grpId}, \tau, (\mathsf{id}_s, \text{'}\mathit{add}\text{'-}\mathsf{id}))$
$v \leftarrow \gamma.\tau.\mathsf{leafof}(\mathsf{id})$
try $(\mathsf{dk}, \mathsf{sk}, \mathsf{dk}') \leftarrow$ query GetSk$((v.\mathsf{ek}, v.\mathsf{vk}, \mathsf{ek}'))$ to $\mathcal{F}_{\mathrm{KS}}$
$(v.\mathsf{dk}, v.\mathsf{sk}) \leftarrow (\mathsf{dk}, \mathsf{sk})$
$\gamma \leftarrow$ `*set-tree-hash`(γ)
try $(\gamma, \mathsf{confTag}) \leftarrow$ `*get-secrets`$(\gamma, \mathsf{dk}', \mathit{ctxt}_1, \mathit{ctxt}_2, \mathsf{id}_s)$
return $(\gamma.\tau.\mathsf{roster}(), \mathsf{id}_s)$

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

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

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

helper `*AKS-kgen()`
$(\mathsf{ek}, \mathsf{dk}) \leftarrow \underline{\mathsf{mmPKE.KG}}()$
$(\mathsf{vk}, \mathsf{sk}) \leftarrow \mathsf{Sig.KG}()$
$(\mathsf{ek}', \mathsf{dk}') \leftarrow \underline{\mathsf{mmPKE.KG}}()$
return $((\mathsf{ek}, \mathsf{vk}, \mathsf{ek}'), (\mathsf{dk}, \mathsf{sk}, \mathsf{dk}'))$

Figure 14: The algorithms of SAIK.

SAIK: Creating epochs

```
helper *create-epoch(γ, id, act)
  γ' ← γ.clone()
  // Apply the action to the tree. Fails if the action is not allowed.
  try γ' ← *apply-act(γ', id, act)
  // Re-key the direct path.
  directPath ← γ'.τ.directPath(γ'.leaf)
  pathSecs[*] ← ⊥
  pathSecs[1] ←$ {0,1}^κ
  for i = 1 to len(directPath) − 1 do
    v ← directPath[i]
    r ← HKDF.Exp(pathSecs[i], 'node')
    (v.ek, v.dk) ← mmPKE.KG(r)
    pathSecs[i + 1] ← HKDF.Exp(pathSec[i], 'path')
  γ'.τ.mergeLvs(γ'.leaf)
  // Re-key the leaf.
  (γ'.leaf.ek, γ'.leaf.dk) ← mmPKE.KG()
  (γ'.leaf.vk, γ'.leaf.sk) ← Sig.KG()
  // Set all context variables and then derive epoch secrets.
  γ'.lastAct ← (id, act)
  γ' ← *set-tree-hash(γ')
  commitSec ← pathSecs[len(pathSecs)]
  (γ', joinerSec) ← *derive-keys(γ', commitSec)
  return (γ', pathSecs, joinerSec)

helper *apply-act(γ', id_s, act)
  req id_s ∈ γ'.τ.roster()
  if act = 'rem'-id_t then
    req id_s ≠ id_t ∧ id_t ∈ γ'.τ.roster()
    γ'.τ.blankPath(γ'.τ.leafof(id_t))
    γ'.τ.leafof(id_t).blank()
  else if act = 'add'-id_t-(ek_t, vk_t) then
    req id_t ∉ γ'.τ.roster()
    v ← γ'.τ.getLeaf()
    (v.id, v.ek, v.vk) ← (id_t, ek_t, vk_t)
    γ.τ.unmerge(v)

helper *transition(γ', ctxt, updEKs', id_s, act)
  // Set keys on the re-keyed path.
  v_s ← γ'.τ.leafof(id_s)
  directPath ← γ'.τ.directPath(v_s)
  (v_s.ek, v_s.vk) ← updEKs'[1]
  i ← 1
  while directPath[i] ∉ {γ'.τ.lca(γ'.leaf, v_s), γ'.τ.root} do
    // If message contains too few ek's, reject it.
    req i + 1 ≤ len(updEKs')
    directPath[i].ek ← updEKs'[i + 1]
    i++
  // Decrypt the path secret using the updated tree.
  try pathSec ← *decrypt-path-secret(γ', id_s, ctxt)
  while i < len(directPath) do
    v ← directPath[i]
    r ← HKDF.Exp(pathSecs[i], 'node')
    (v.ek, v.dk) ← mmPKE.KG(r)
    pathSec ← HKDF.Exp(pathSec, 'path')
    i++
  commitSec ← pathSec
  γ'.τ.mergeLvs(v_s)
  // Set all context variables and then derive epoch secrets.
  γ'.lastAct ← (id_s, act)
  γ' ← *set-tree-hash(γ')
  (γ', joinerSec) ← *derive-keys(γ', commitSec)
  return γ'

helper *get-secrets(γ', dk', ctxt_1, ctxt_2, id_s)
  try pathSec ← mmPKE.Dec(dk, ctxt_1)
  try joinerSec ← mmPKE.Dec(dk, ctxt_2)
  v ← γ'.τ.lca(γ'.leaf, γ'.τ.leafof(id_s))
  while v ≠ γ'.τ.root do
    r ← HKDF.Exp(pathSec, 'node')
    (ek, v.dk) ← mmPKE.KG(r)
    req v.ek = ek
    pathSec ← HKDF.Exp(pathSec, 'path')
    v ← v.parent
  γ' ← *derive-epoch-keys(γ', joinerSec)
  return γ'
```

SAIK: Key schedule

```
helper *derive-keys(γ, γ', commitSec)
  joinerSec ← HKDF.Ext(γ.initSec, commitSec)
  γ' ← *derive-epoch-keys(γ', joinerSec)
  return γ', joinerSec

helper *derive-epoch-keys(γ', joinerSec)
  epSec ← HKDF.Ext(joinerSec, γ'.grpCtxt())
  γ'.appSec ← HKDF.Exp(epSec, 'app')
  γ'.membKey ← HKDF.Exp(epSec, 'membership')
  γ'.initSec ← HKDF.Exp(epSec, 'init')
  γ'.confTag ← HKDF.Exp(epSec, 'confirmation')
  return γ'
```

SAIK: Tree hash

```
helper *set-tree-hash(γ')
  γ'.treeHash ← *tree-hash(γ'.τ.root)
  return γ'

helper *tree-hash(v)
  if v.isleaf then
    return Hash(v.nodeIdx, v.ek, v.vk)
  else
    ℓ ← len(v.children)
    for i ∈ [ℓ] do h_i ← *tree-hash(v.children[i])
    h ← (h_1, ..., h_ℓ)
    return Hash(v.nodeIdx, v.ek, v.unmLvs, h)
```

Figure 15: Additional helper methods for SAIK.

Game mmOW-RCCA

$\underline{\mathrm{Exp}^{\mathsf{mmIND\text{-}RCCA}}_{\mathsf{mmPKE},N}(\mathcal{A})}$

$(\mathcal{A}_1, \mathcal{A}_2) \leftarrow \mathcal{A}$
for $i \in [N]$ **do** $(\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{mmPKE.KG}()$
$\mathsf{Corr} \leftarrow \emptyset$
$(\vec{\mathsf{ek}}, \vec{m}, S, st) \leftarrow \mathcal{A}_1^{\mathbf{Dec}_1,\mathbf{Cor}}(\mathsf{ek}_1, \ldots, \mathsf{ek}_N)$
req $|\vec{m}| = |\vec{\mathsf{ek}}| \wedge S \subseteq [|\vec{m}|]$
$EK^* \leftarrow \{\vec{\mathsf{ek}}[j] : j \in S\}$
$m^* \xleftarrow{\$} \mathcal{M}$
for $j \in S$ **do** $\vec{m}[j] \leftarrow m^*$
$m' \leftarrow \mathcal{A}^{\mathbf{Dec}_2,\mathbf{Cor}}(\mathsf{mmPKE.Enc}(\vec{\mathsf{ek}}, \vec{m}), st)$
req $EK^* \subseteq \{\mathsf{ek}_i : i \in [N] \setminus \mathsf{Corr}\}$
return $m^* = m'$

Oracle $\underline{\mathbf{Dec}_1(i, c)}$

req $i \in [N]$
return $\mathsf{Dec}(\vec{\mathsf{dk}}_i, c)$

Oracle $\underline{\mathbf{Cor}(i)}$

req $i \in [N]$
$\mathsf{Corr} \mathrel{+}\!\!\leftarrow i$
return dk_i

Oracle $\underline{\mathbf{Dec}_2(i, c)}$

req $i \in [N]$
$m \leftarrow \mathsf{mmPKE.Dec}(\vec{\mathsf{dk}}_i, c)$
if $\mathsf{ek}_i \in EK^* \wedge m = m^*$ **then**
  return *'test'*
else return m

Figure 16: Experiment defining mmOW-RCCA **security of mmPKE schemes.**

Theorem G.2. *Let* mmPKE *be an mmPKE scheme with message space $\mathcal{M}$. For any adversary $\mathcal{A}$, there exists an adversary $\mathcal{B}$ such that*

$$\mathrm{Adv}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE},N}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{mmIND\text{-}RCCA}}_{\mathsf{mmPKE},N}(\mathcal{B}) + \frac{2}{\mathcal{M}}.$$

Proof. The proof closely follows the typical proofs showing that IND security implies OW security for standard encryption.

Given an adversary $\mathcal{A}$ against mmOW-RCCA security, the reduction $\mathcal{B}$ attacking mmIND-RCCA simply runs $\mathcal{A}$ on the public keys it receives in the mmIND-RCCA experiment and forwards all $\mathcal{A}$'s oracle queries to its mmIND-RCCA oracles. When $\mathcal{A}$ outputs the triple $(\vec{\mathsf{ek}}, \vec{m}, S)$, $\mathcal{B}$ computes the challenge ciphertext as follows. First, it initializes $\vec{m}^*_0, \vec{m}^*_1 \leftarrow \vec{m}$. Then, it picks two random messages m^*_0 and m^*_1 and for each $j \in S$ sets $\vec{m}^*_0[j] \leftarrow m^*_0$ and $\vec{m}^*_1[j] \leftarrow m^*_1$. It sends $\vec{\mathsf{ek}}$ together with $\vec{m}^*_0$ and $\vec{m}^*_1$ to the mmIND-RCCA experiment, receives the challenge ciphertext c^* and sends it to $\mathcal{A}$. At the end of the experiment, $\mathcal{A}$ outputs a guess m'. If $m' = m^*_1$, then $\mathcal{B}$ outputs 1. Else, it outputs 0.

First, it is easy to see that if $\mathcal{A}$ does not violate any **req** statements in the emulation, then $\mathcal{B}$ does not violate any **req** statements in the mmIND-RCCA game. In particular, $\vec{m}^*_0$ and $\vec{m}^*_1$ clearly have the same leakage. It is also easy to see that if $\mathcal{A}$ does not trivially win by corruptions then $\mathcal{B}$ does not either.

Second, observe that if $\mathcal{B}$'s challenger uses the bit $b = 1$, then $\mathcal{B}$ emulates $\mathcal{A}$'s experiment perfectly, unless $\mathcal{A}$ inputs to $\mathbf{Dec}_2$ something that decrypts to m^*_0. The reason is that in this case $\mathcal{B}$ replies with *'test'* (forwarded from its oracle), while $\mathcal{A}$ should receive m^*_0. Since m^*_0 is random and independent of $\mathcal{A}$'s view, this happens with probability at most $1/\mathcal{M}$. Therefore, it is easy to see that

$$\Pr\left[\mathrm{Exp}^{\mathsf{mmIND\text{-}RCCA}}_{\mathsf{mmPKE},N,n,1}(\mathcal{B}) \Rightarrow 1\right] \leq \mathrm{Adv}^{\mathsf{mmIND\text{-}RCCA}}_{\mathsf{mmPKE},N,n}(\mathcal{A}) + \frac{1}{\mathcal{M}}.$$

If $\mathcal{B}$ is in the experiment with the bit $b = 0$, then m^*_1 is independent of $\mathcal{A}$'s view, so the probability that it outputs $m' = m^*_1$ and hence also that $\mathcal{B}$ outputs 1 is at most $\frac{1}{\mathcal{M}}$. □

H SECURITY OF SAIK

The security predicates for SAIK are defined in Fig. 17. See Sec. 6 for the intuition. The stronger version of the predicates that is not achieved by SAIK skips the code in $\boxed{\text{boxes,}}$ while the weaker version includes the whole code. In Sec. 8.1 we sketch how to modify SAIK to achieve the stronger version.

For the mmPKE scheme we assume a security property called mmOW-RCCA, defined in App. G. The notion is strictly weaker than mmIND-CCA; in App. G we prove the implication.

Theorem H.1. *Let $\mathcal{F}_{\mathrm{CGKA}}$ be the CGKA functionality with predicates* **confidential** *and* **authentic** *defined in Fig. 17. Let* SAIK *be instantiated with schemes* mmPKE, Sig *and* MAC*, and with the* HKDF *functions modelled as a random oracle* Hash*. Let $\mathcal{A}$ be any environment. Denote the output of $\mathcal{A}$ from the real execution with* SAIK *and the hybrid functionality $\mathcal{F}_{\mathrm{AKS}}$ from Fig. 13 as $\textsc{real}_{\mathsf{SAIK},\mathcal{F}_{\mathrm{AKS}}}(\mathcal{A})$ and the output of $\mathcal{A}$ from the ideal execution with $\mathcal{F}_{\mathrm{CGKA}}$ and a simulator $\mathcal{S}$ as $\textsc{ideal}_{\mathcal{F}_{\mathrm{CGKA}},\mathcal{S}}(\mathcal{A})$. There exists a simulator $\mathcal{S}$ and adversaries $\mathcal{B}_1$ to $\mathcal{B}_4$ such that*

$$\begin{aligned}\Pr[\textsc{ideal}_{\mathcal{F}_{\mathrm{CGKA}},\mathcal{S}}(\mathcal{A}) = 1] - \Pr\left[\textsc{real}_{\mathsf{SAIK},\mathcal{F}_{\mathrm{AKS}}}(\mathcal{A}) = 1\right] \leq \\ \mathrm{Adv}^{\mathsf{CR}}_{\mathsf{Hash}}(\mathcal{B}_1) \\ + q_e^2(q_e + 1)\log(q_n) \cdot \mathrm{Adv}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE},q_e \log(q_n),q_n}(\mathcal{B}_2) \\ + 2q_e \cdot \mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig}}(\mathcal{B}_3) \\ + q_e \cdot \mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{B}_4) + 3q_h q_e^2(q_e + 1)/2^{\kappa},\end{aligned}$$

where q_e, q_n and q_h denote bounds on the number of epochs, the group size and the number of $\mathcal{A}$'s queries to the random oracle modelling the Hash*, respectively.*

H.1 Proof Outline

Int the remaining subsections we prove Theorem H.1.

The proof proceeds in a sequence of hybrids, transitioning from the real to the ideal world. Hybrid 1 differs from the real world only syntactically. That is, the environment $\mathcal{A}$ interacts with a dummy CGKA functionality $\mathcal{F}^1_{\mathrm{CGKA}}$ which allows the simulator to set all outputs. This means that $\mathcal{F}^1_{\mathrm{CGKA}}$ gives no security guarantees.

Security predicates for SAIK

$\textbf{confidential}(\text{epid}) \iff \neg\texttt{*in-det-tree}(\text{epid}) \wedge \texttt{*grp-secs-secure}(\text{epid})$
$\textbf{authentic}(\text{epid}, \text{id}) \iff \neg\texttt{*in-det-tree}(\text{epid}) \wedge \big(\text{epid} = 0 \vee \texttt{*grp-secs-secure}(\text{epid}) \vee \texttt{*ind-secs-secure}(\text{epid}, \text{id})\big)$

$\texttt{*in-det-tree}(\text{epid}) \iff \neg^{*}\text{ancestor}(0, \text{epid})$

$\texttt{*grp-secs-secure}(\text{epid} = 0) \iff \text{HG}[\text{epid}].\text{exp} = \varnothing$
$\texttt{*grp-secs-secure}(\text{epid} > 0) \iff \text{HG}[\text{epid}].\text{exp} = \varnothing \wedge \neg\text{HG}[\text{epid}].\text{inj} \wedge \big(\texttt{*grp-secs-secure}(\text{HG}[\text{epid}].\text{par}) \vee \texttt{*all-ind-secs-secure}(\text{epid})\big)$

$\texttt{*all-ind-secs-secure}(\text{epid}) \iff \forall \text{id} \in \text{HG}[\text{epid}].\text{mem} \setminus \{\text{HG}[\text{epid}].\text{sndr}\} : \texttt{*ind-secs-secure}(\text{HG}[\text{epid}].\text{par}, \text{id})$

$\texttt{*ind-secs-secure}(\text{epid}, \text{id}) \iff \big(\nexists \text{epid}' : \texttt{*share-ind-secs}(\text{epid}, \text{epid}', \text{id}) \wedge \texttt{*ind-secs-bad}(\text{epid}', \text{id})\big) \boxed{\wedge\ \neg\texttt{*exposed-ind-secs-weak}(\text{epid}, \text{id})}$

$\texttt{*share-ind-secs}(\text{epid}, \text{epid}', \text{id}) \iff$ epid and epid′ are the same or connected via undirected path of epochs epid′′ such that $\text{HG}[\text{epid}''].\text{sndr} \neq \text{id} \wedge \text{HG}[\text{epid}].\text{act} \notin \{\textit{'rem'}\text{-id}, \textit{'add'}\text{-id}\}$

$\texttt{*ind-secs-bad}(\text{epid}, \text{id}) \iff \text{id} \in \text{HG}[\text{epid}].\text{exp} \vee (\text{HG}[\text{epid}].\text{sndr} = \text{id} \wedge \text{HG}[\text{epid}].\text{inj}) \vee (\text{HG}[\text{epid}].\text{act} = \textit{'add'}\text{-id} \wedge \text{HG}[\text{epid}].\text{inj})$

$\texttt{*exposed-ind-secs-weak}(\text{epid}, \text{id}) \iff \exists \text{epid}_1, \text{epid}_2, \text{epid}_3$: all of the following conditions are satisfied:
(1) $\text{epid}_1 \neq \text{epid}_2 \wedge {}^{*}\text{ancestor}(\text{epid}_1, \text{epid}) \wedge {}^{*}\text{ancestor}(\text{epid}_2, \text{epid}_3)$
(2) $\text{HG}[\text{epid}_1].\text{act} = \text{HG}[\text{epid}_2].\text{act} = \textit{'add'}\text{-id}$
(3) $\texttt{*share-ind-secs}(\text{epid}_1, \text{epid}, \text{id}) \wedge \texttt{*share-ind-secs}(\text{epid}_2, \text{epid}_3, \text{id})$
(4) $\text{HG}[\text{epid}_2].\text{inj} \wedge \text{id} \in \text{HG}[\text{epid}_3].\text{exp}$

Figure 17: Security predicates instantiating $\mathcal{F}_{\text{CGKA}}$ constructed by SAIK**.**

The next three hybrids introduce the guarantees of consistency, confidentiality and authenticity, one by one. More precisely, in hybrid 2, $\mathcal{A}$ interacts with $\mathcal{F}^2_{\text{CGKA}}$ which is the same as $\mathcal{F}_{\text{CGKA}}$, except it uses **confidential** and **authentic** set to false. In particular, this means that $\mathcal{F}^2_{\text{CGKA}}$ builds a history graph, enforces its consistency and uses it to compute outputs. In hybrid 3, $\mathcal{A}$ interacts with $\mathcal{F}^3_{\text{CGKA}}$ which uses the original **confidential** predicate, and in hybrid 4 it interacts with $\mathcal{F}^4_{\text{CGKA}}$ which also uses the original **authentic** predicate. Notice that $\mathcal{F}^4_{\text{CGKA}}$ is $\mathcal{F}_{\text{CGKA}}$.

In the next subsections, we define the hybrids and show that each pair of consecutive hybrids is indistinguishable for $\mathcal{A}$. Intuitively, each such statement means that SAIK provides the introduced security guarantee.

Hybrid 1. This is the experiment $\text{IDEAL}_{\mathcal{F}^1_{\text{CGKA}}, \mathcal{S}^1}$ where the dummy functionality $\mathcal{F}^1_{\text{CGKA}}$ sends all inputs to the simulator $\mathcal{S}^1$ and allows it to set all outputs. $\mathcal{S}^1$ executes SAIK.

H.2 SAIK Guarantees Consistency

The following hybrid introduces consistency.

Hybrid 2: $\text{IDEAL}_{\mathcal{F}^2_{\text{CGKA}}, \mathcal{S}^2}$. The functionality $\mathcal{F}^2_{\text{CGKA}}$ is the same as $\mathcal{F}_{\text{CGKA}}$ except it uses **confidential** = **authentic** = `false`. The simulator $\mathcal{S}^2$ is described later in this section.

In the reminder of this section, we construct the simulator $\mathcal{S}^2$ and show that hybrids 1 and 2 are indistinguishable.

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

$$\left| \begin{array}{l} \Pr\left[\text{IDEAL}_{\mathcal{F}^2_{\text{CGKA}}, \mathcal{S}^2}(\mathcal{A}) \Rightarrow 1\right] \\ -\Pr\left[\text{IDEAL}_{\mathcal{F}^1_{\text{CGKA}}, \mathcal{S}^1}(\mathcal{A}) \Rightarrow 1\right] \end{array} \right| \leq \text{Adv}^{\text{CR}}_{\text{Hash}}(\mathcal{B}) + q_e/2^{\kappa},$$

where Hash *models the* HKDF.Exp *and* HKDF.Ext *functions and q_e denotes an upper bound on the number of epochs.*

The simulator. We first describe $\mathcal{S}^2$. In general, it runs SAIK just like $\mathcal{S}^1$, only its interaction with the functionality is different. Most importantly, $\mathcal{F}^2_{\text{CGKA}}$ requires that $\mathcal{S}^2$ identifies epochs into which parties transition. Doing this correctly is crucial for proving that SAIK guarantees consistency, because $\mathcal{F}^2_{\text{CGKA}}$ enforces it by computing outputs and asserting conditions relative to parties' current epochs. (It must also be done so that we can later prove that SAIK guarantees confidentiality and authenticity.)

$\mathcal{S}^2$ identifies epochs by their epoch secrets epSec, computed by SAIK on Receive and Send. Recall that a party id transitioning from an epoch $E[1]$ to E_2 computes E_2's epSec by hashing $E[1]$'s init secret, the new commit secret (combined into the joiner) generated by E_2's creator and E_2's context. We will show that these values contain enough information for epSec to *uniquely* identify E_2. Recall also that the group and init key of E_2 are derived from epSec. The simulator is described in more detail in Fig. 18.

Proof. We next prove Theorem H.2. Observe that hybrids 1 and 2 are identical unless one of the following two events occurs in hybrid 2:

BreaksCons : Either the output of a party on Receive or Key computed according to $\mathcal{F}^2_{\text{CGKA}}$ and $\mathcal{S}^2$ is different than the output $\mathcal{S}^1$ would compute according to SAIK in hybrid 1, or an **assert** condition is false.

EpidColl : An honestly created epoch has the same epSec as an existing epoch.

Observe that since an honest sender mixes a fresh commitSec into the derivation of epSec, the probability of EpidColl is at most $q_e/2^{\kappa}$ (where κ is the length of all secrets). It remains to show that if $\mathcal{A}$

Simulator $\mathcal{S}^2$

$\mathcal{S}^2$ keeps a list EpSecs, where EpSecs[epid] stores the epoch secret identifying epoch epid. It runs SAIK and interacts with $\mathcal{F}^2_{\text{CGKA}}$ as follows:

- If SAIK outputs $\perp$ on Send or Receive, $\mathcal{S}^2$ sends *ack* set to false.
- On each Send, $\mathcal{S}^2$ computes the new epoch's epSec and appends it to EpSecs. It sends to $\mathcal{F}_{\text{CGKA}}$ the message C computed using SAIK.
- On each Receive, $\mathcal{S}^2$ first sends to $\mathcal{F}_{\text{CGKA}}$ the values sndr′, act′ from the message. If the receiver is not removed, $\mathcal{S}^2$ sends epid into which id transitions chosen as follows:
  - If there is a epid s.t. EpSecs[epid] = epSec, then $\mathcal{S}^2$ sends this (unique) epid to $\mathcal{F}^2_{\text{CGKA}}$.
  - Else, $\mathcal{S}^2$ appends epid to EpSecs and sends epid = $\perp$ to $\mathcal{F}^2_{\text{CGKA}}$.

  Finally, if a detached root is created and $\mathcal{F}^2_{\text{CGKA}}$ asks for the member set mem′, $\mathcal{S}^2$ computes it from the new member's ratchet tree.

Figure 18: The simulator for the proof of the security of SAIK**.**

triggers BreaksCons, then a reduction $\mathcal{B}$ can extract from a hash collision. (Theorem H.2 follows by the standard difference lemma.)

Let $\mathcal{A}$ be any environment and assume that at the end of hybrid 2 with $\mathcal{A}$ there are no hash between values hashed by $\mathcal{S}^2$ while running SAIK on behalf of honest parties. We show that in this case BreaksCons cannot occur. This proves the claim, because if there was a hash collision between values hashed by honest parties, then $\mathcal{B}$ could extract them by emulating $\mathcal{S}^1$.

Observe that if two parties transition to the same epoch epid, then by definition of $\mathcal{S}^2$ they compute the same epSec. Recall that they compute epSec $\leftarrow$ Hash(joinerSec, grpCtxt) (Fig. 15), where grpCtxt = (grpId, treeHash, id_s-act) (??). Since there are no hash collisions, this means that the parties also agree on the following values:

a) The creator HG_s of epid, the action act it performed and the public part of the ratchet tree, included in treeHash. This implies agreement on the roster, which is encoded in the tree leaves.
b) The group key in epid, derived as Hash(epSec, *'app'*).

Moreover, let epSec′ denote the epoch secret of epid's parent. We have joinerSec = Hash(initSec′, commitSec), where initSec′ = Hash(epSec′, *'init'*) and commitSec is freshly chosen for epid by its creator. Therefore, parties in epid also agree on:

c) The parent epoch epid′ identified by epSec′.

Observe that the check Hash(initSec′, commitSec) = joinerSec is verified by current members transitioning to epid but not by joiners. However, joiners implicitly agree with current members on the parent epid′. That is, if an id_r joins into epid, then epid has parent epid′ (unknown to id_r) or no parent at all (for detached roots).

We next show that agreement on a), b) and c) implies that BreaksCons does not occur. First, a) and b) imply that all parties joining an epoch epid output the same value, all parties transitioning there output the same, and afterwards all output the same key. Second, c) implies that HG is a forest, i.e., each epoch has one parent.

Third, we have to argue that parties' outputs are the same as computed by $\mathcal{F}^2_{\text{CGKA}}$. This is obvious for the key (always chosen by $\mathcal{S}^2$ to match), sender and action. For the member set, we will show that the ratchet tree of parties in an epoch epid is consistent with HG[epid].mem computed by $\mathcal{F}^2_{\text{CGKA}}$. We use induction on the distance of epid to the root. If epid is the main root, then the statement is true by definition and if it is a detached roots, then $\mathcal{S}^2$ chooses mem to match the ratchet tree. For any non-root epid, some party id must have transitioned there from its parent epid′ (on Receive or Send). By induction hypothesis, the ratchet tree in epid′ is consistent with HG[epid].mem. By agreement on act in a), id modifies the tree the same way as $\mathcal{F}^2_{\text{CGKA}}$ modifies HG[epid].mem, which proves the statement.

H.3 SAIK Guarantees Confidentiality

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

Hybrid 3: $\text{IDEAL}_{\mathcal{F}^3_{\text{CGKA}},\mathcal{S}^3}$. The functionaliy $\mathcal{F}^3_{\text{CGKA}}$ uses the original **confidential** predicate from $\mathcal{F}_{\text{CGKA}}$. The simulator $\mathcal{S}^3$ is the same as $\mathcal{S}^2$.

In the remainder of this section, we show that if mmPKE is mmOW-RCCA secure, then SAIK guarantees confidentiality, that is, that hybrids 2 and 3 are indistinguishable. Formally, we prove the following theorem.

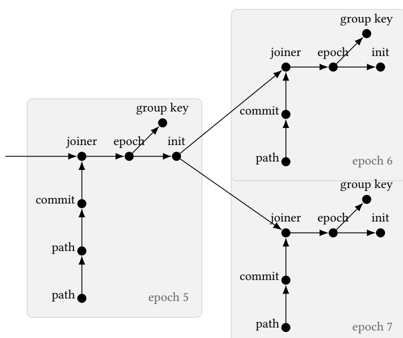
Theorem H.3. *For any environment $\mathcal{A}$, there exists an adversary $\mathcal{B}$ such that*

$$\begin{aligned}\Pr\left[\text{IDEAL}_{\mathcal{F}^3_{\text{CGKA}},\mathcal{S}^3}(\mathcal{A}) \Rightarrow 1\right] - \Pr\left[\text{IDEAL}_{\mathcal{F}^2_{\text{CGKA}},\mathcal{S}^2}(\mathcal{A}) \Rightarrow 1\right] \\ \leq 4q_e^2 q_h/2^\kappa \\ + q_e^2 \log(q_n) \cdot \text{Adv}^{\text{mmOW-RCCA}}_{\text{mmPKE},q_e \log(q_n),q_n}(\mathcal{B}),\end{aligned}$$

where the HKDF functions are modeled as a random oracle and where q_n, q_h and q_e are upper bounds on, respectively, the group size, the number of $\mathcal{A}$'s hash queries and the number of epochs.

Game-based perspective. For better intuition, observe that hybrids 2 and 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 2 (technically, computed by the simulator according to SAIK) and random and independent in hybrid 3 (technically, sampled by $\mathcal{F}^3_{\text{CGKA}}$). This means that distinguishing between hybrids 2 and 3 can 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.

Key Graphs. A key graph visualizes different secrets created in an execution of SAIK and hash relations between them. Each node in the graph corresponds to a secret, e.g. the group key in epoch 5, and has assigned its value. The directed edges are interpreted as follows: the value of a node is the hash of the values of all its in-neighbors with an appropriate label. If a node has many out-neighbors, then the value of each out-neighbor is computed by hashing with a different label (i.e., the values of out-neighbors are domain-separated). Values of source nodes are either chosen at random by the protocol or injected by the adversary. The key-graph nodes are partitioned by epochs: Secrets of an epoch epid are those created when epid is created. We distinguish two types of secrets: *group secrets* which include the init, joiner and epoch secrets as well as the group key, and *individual secrets*, which include path secrets, the last being the commit secret. An example key graph is given below. We removed membership secrets for simplicity. Note that the epochs 6 and 7 are created in parallel, that is, we have a group fork.



Note that in case of injections the values of nodes may not be unique. However, the values of epoch secrets uniquely identify epochs. Note also that the values of group secrets of epid appear only in the states of parties in epid. On the other hand, mmPKE keys derived from path secrets of epid appear in ratchet trees stored by parties in multiple epochs.

Bad events. Let $\mathcal{A}$ be any environment. The goal is to show that $\mathcal{A}$ cannot distinguish the real group keys of confidential epochs it sees in hybrid 2 from random and independent keys in hybrid 3. Since epochs in detached trees are not confidential, in the remainder of the proof we only consider epochs in the main history-graph tree.

Observe that there are only two dependencies between the real group key appSec of an epoch epid and the rest of the experiment: appSec is stored by parties in epid and it is the hash of epid's unique epoch secret epSec. If epid is confidential, then no party in epid, i.e., no party storing appSec is corrupted. Therefore, unless $\mathcal{A}$ inputs epSec to the RO, the real group key is independent of the rest of the experiment. In other words, unless $\mathcal{A}$ inputs epSec to the RO, the real group key outputted in hybrid 2 is distributed identically as the random key in hybrid 3.

Therefore, $\mathcal{A}$'s distinguishing advantage is upper-bounded by the probability that the following event $\mathsf{SecsHashed}_{\mathsf{epid}}$ occurs for at least one epoch epid. For convenience, the event is more general and also considers init and joiner secrets.

Event $\mathsf{SecsHashed}_{\mathsf{epid}}$ **:** At the end of the experiment, epid is confidential and epid's init, epoch or joiner secret is contained in a value inputted by $\mathcal{A}$ to RO.

Formally, it is left to prove the following lemma.

Lemma H.4. *There exists a reduction $\mathcal{B}$ such that*

$$\Pr[\exists \mathsf{epid} : \mathsf{SecsHashed}_{\mathsf{epid}}] \le 4q_e^2 q_h/2^\kappa + q_e^2 \log(q_n) \cdot \mathrm{Adv}^{\mathsf{mmIND\text{-}RCCA}}_{\mathsf{mmPKE},q_e \log(q_n),q_n}(\mathcal{B}).$$

Bounding probability of bad events. An epoch epid is confidential if *grp-secs-secure(epid) is true (all predicates are defined in Fig. 17). The latter predicate is recursive, starting at the root epoch with epid = 0. Accordingly, we will prove a recursive upper bound on the probability of $\mathsf{SecsHashed}_{\mathsf{epid}}$. Formally, Theorem H.4 is implied by the following lemma.

Lemma H.5. *There exists a reduction $\mathcal{B}$ and (arbitrary) events* $\mathsf{BreaksRCCA}_{\mathsf{epid}}$[9] *for* $\mathsf{epid} \in \mathbb{N}$ *such that*

a) $\Pr[\exists \mathsf{epid} : \mathsf{SecsHashed}_0] \le 4q_h/2^\kappa$.

b) For each $\mathsf{epid} > 0$ *with parent* epid_p*, we have*
$$\Pr[\mathsf{SecsHashed}_{\mathsf{epid}}] \le 4q_h/2^\kappa + \Pr[\mathsf{SecsHashed}_{\mathsf{epid}_p}] + \Pr[\mathsf{BreaksRCCA}_{\mathsf{epid}}].$$

c) $$\Pr[\exists \mathsf{epid} : \mathsf{BreaksRCCA}_{\mathsf{epid}}] \le q_e \log(q_n) \cdot \mathrm{Adv}^{\mathsf{mmIND\text{-}RCCA}}_{\mathsf{mmPKE},q_e \log(q_n),q_n}(\mathcal{B}).$$

To see that Theorem H.5 implies Theorem H.4, observe that Theorem H.5 implies

$$\begin{aligned}
&\Pr[\mathsf{SecsHashed}_{\mathsf{epid}}] \\
&\quad \le \sum_{i=0}^{\mathsf{epid}-1} (4q_h/2^\kappa + \Pr[\mathsf{BreaksRCCA}_i]) \\
&\quad \le 4q_e q_h/2^\kappa + \Pr[\exists \mathsf{epid} : \mathsf{BreaksRCCA}_{\mathsf{epid}}] \\
&\quad \le 4q_e q_h/2^\kappa + q_e \log(q_n) \cdot \mathrm{Adv}^{\mathsf{mmIND\text{-}RCCA}}_{\mathsf{mmPKE},q_e \log(q_n),q_n}(\mathcal{B}),
\end{aligned}$$

where the first step follows from a) and b) in Theorem H.5. We arrive at Theorem H.4, since by union bound $\Pr[\exists \mathsf{epid} : \mathsf{SecsHashed}_{\mathsf{epid}}] \le q_e \cdot \max_{\mathsf{epid}} \Pr[\mathsf{SecsHashed}_{\mathsf{epid}}]$.

Proof of Theorem H.5 a). The root epoch does not have joiner and epoch secrets. The init secret of epoch 0 is chosen at random by the group creator $\mathsf{id}_{\mathsf{creator}}$. Moreover, it is independent of the rest of the experiment apart from being stored by $\mathsf{id}_{\mathsf{creator}}$ in epoch 0. The reason is that any other values are derived by first hashing it, and outputs of the RO are independent of the inputs. If epoch 0 is confidential, then $\mathsf{id}_{\mathsf{creator}}$ is not corrupted in epoch 0, so $\mathcal{A}$ has no information about the init secret. Therefore, the best strategy for $\mathcal{A}$ to trigger $\mathsf{SecsHashed}_0$ is by guessing the init secret, which succeeds with probability at most $q_h/2^\kappa < 4q_h/2^\kappa$.

[9] The lemma implies Theorem H.4 no matter what $\mathsf{BreaksRCCA}_{\mathsf{epid}}$ is. The name will become clear later in the proof.

Proof of Theorem H.5 b). Take any non-root epoch $\mathsf{epid} > 0$ with parent epid_p. Let $\mathsf{initSec}$, epSec, $\mathsf{joinerSec}$ and $\mathsf{commitSec}$ denote epid's init, epoch, joiner and commit secrets. Let $\mathsf{initSec}_p$ denote epid_p's init secret.

Observe that the only dependencies between initSec, epSec, joinerSec and the rest of the experiment are as follows: 1) initSec is stored by parties in epid, 2) joinerSec is the output of the RO on input commitSec together with $\mathsf{initSec}_p$, 3) joinerSec is encrypted to new members. (Note that any other values are derived by first hashing it, and outputs of the RO are independent of the inputs.)

Assume for a moment that epid is confidential. Then, no party in epid is corrupted, so dependency 1) does not exist. Recall that confidentiality requires either `*grp-secs-secure`(epid_p) is true or `*all-ind-secs-secure`(epid) is true. Observe that dependency 2) does not exist either unless one of the following events occurs:

Event $\mathsf{InitHashed}_{\mathsf{epid}_p}$ **:** At the end of the experiment, the value of the predicate `*grp-secs-secure`(epid_p) is true and $\mathsf{initSec}_p$ (of epid_p) is contained in some value inputted by $\mathcal{A}$ to the RO.

Event $\mathsf{CommHashed}_{\mathsf{epid}}$ **:** At the end of the experiment, the predicate `*all-ind-secs-secure`(epid) is true and commitSec (of epid) is contained in some value inputted by $\mathcal{A}$ to RO.

This means that unless $\mathsf{InitHashed}_{\mathsf{epid}_p}$ or $\mathsf{CommHashed}_{\mathsf{epid}}$ occurs, $\mathcal{A}$ has no information about initSec and epSec. Therefore, the best strategy for $\mathcal{A}$ to trigger $\mathsf{SecsHashed}_{\mathsf{epid}}$ is to either guess initSec or epSec at random, or trigger one of the above events, or input the joiner secret based only on dependency 3). We capture the last event by

Event $\mathsf{joinerSec}_{\mathsf{epid}}$ **:** At the end of the experiment, none of the values inputted by $\mathcal{A}$ to the RO includes $\mathsf{initSec}_p$ (of epid_p) and commitSec (of epid) together, but some value contains joinerSec (of epid).

Therefore, we have

$$\Pr\left[\mathsf{SecsHashed}_{\mathsf{epid}}\right] \leq 2q_h/2^{\kappa} + \Pr[\mathsf{InitHashed}_{\mathsf{epid}_p}] + \Pr\left[\mathsf{CommHashed}_{\mathsf{epid}}\right].$$

By definition, $\Pr[\mathsf{InitHashed}_{\mathsf{epid}_p}] \leq \Pr[\mathsf{SecsHashed}_{\mathsf{epid}_p}]$. Moreover, we define

Event $\mathsf{BreaksRCCA}_{\mathsf{epid}}$ **:** Either $\mathsf{CommHashed}_{\mathsf{epid}}$ or $\mathsf{JoinHashed}_{\mathsf{epid}}$ occurs.[10]

This proves the claim.

Proof of Theorem H.5 c). We construct two reductions $\mathcal{B}_1$ and $\mathcal{B}_2$ whose advantages bound the probability of the event $\exists\mathsf{epid} : \mathsf{JoinHashed}_{\mathsf{epid}}$ and of $\exists\mathsf{epid} : \mathsf{CommHashed}_{\mathsf{epid}}$, respectively.

Lemma H.6. *There exists a reduction $\mathcal{B}_1$ such that*

$$\Pr(\exists\mathsf{epid} : \mathsf{JoinHashed}_{\mathsf{epid}}) \leq q_e \cdot \mathrm{Adv}^{\mathsf{mmIND\text{-}RCCA}}_{\mathsf{mmPKE},1,q_n}(\mathcal{B}_1).$$

Proof. Take any epoch epid with parent epid_p (the root does not have a joiner secret). Observe that the joiner secret of epid is never stored in the state of SAIK. Moreover, the only message that may include it is the message creating epid which potentially encrypts it to a new member. This means that if $\mathcal{A}$ does not input to the RO the init secret of epid_p together with the commit secret of epid, then the only part of its view that may depend on the joiner of epid is the ciphertext in the message creating epid. In particular, if epid is honestly created and adds a party id_r, then the ciphertext encrypts the joiner under id_r's key from the AKS (i.e., our PKI). Since the AKS is uncorruptible and id_r deletes the secret key immediately after using it, this means that inputting the joiner to the RO implies breaking security of mmPKE.

More formally, consider the following reduction $\mathcal{B}_1$ playing the mmOW-RCCA game with 1 user. $\mathcal{B}_1$ guesses an epoch $\mathsf{epid}^* \in [q_e]$ and runs $\mathcal{A}$, emulating the CGKA functionality and the simulator as in hybrid 2. If epid^* is injected or not created on an add, $\mathcal{B}_1$'s emulation is identical to hybrid 2. Otherwise, $\mathcal{B}_1$'s emulation will be identical to hybrid 2 but where the joiner secret $\mathsf{joinerSec}^*$ of epid^* is replaced the mmOW-RCCA challenge message m^*. $\mathcal{B}_1$ will use a special symbos *'test'* to denote this unknown value of m^* in the emulation.

In particular, when an id_s creates epid^* while adding an id_r, $\mathcal{B}_1$ embeds the single public key ek^* from its game as the key generated for id_r by the AKS (recall that the AKS generates the key pair $(\mathsf{ek}^*, \mathsf{dk}^*)$ at the moment id_s requests it to create the epoch). Further, $\mathcal{B}_1$ computes SAIK's state for epid^* according to the protocol. It then replaces the (fresh) joiner secret generated by SAIK by *'test'* in all places, including the programmed RO inputs and outputs. Finally, $\mathcal{B}_1$ sends to the challenger the message vector $\vec{m}$ encrypted by id_s and the last index in this vector, denoting the (only) position of the joiner secret. The challenger sends back a ciphertext C^*, which $\mathcal{B}_1$ uses in the message sent by id_s.

If id_r uses dk^*, $\mathcal{B}_1$ uses the Dec oracle. Note that Dec may output *'test'*, which is used consistently with the symbol for the unknown joiner m^*. If $\mathcal{A}$ inputs to tje the RO the init secret $\mathsf{initSec}^*$ of epid^*'s parent together with the commit $\mathsf{commitSec}^*$ of epid^*, $\mathcal{B}_1$ halts and gives up. At the end of the experiment, $\mathcal{B}_1$ searches all $\mathcal{A}$'s queries to the RO for an m^* that allows it to win.

We first claim that, until $\mathcal{B}_1$ gives up or the experiment ends, its emulation is perfect. In particular, since $\mathcal{A}$ does not input $\mathsf{initSec}^*$ with $\mathsf{commitSec}^*$ to the RO, which means that, apart from C^*, its experiment is independent of $\mathsf{joinerSec}^*$. This means that $\mathcal{B}_1$ simulates it perfectly by using *'test'* instead of $\mathsf{joinerSec}^*$. Second, we claim that if $\mathsf{JoinHashed}_{\mathsf{epid}^*}$ occurs, then $\mathcal{B}_1$ wins. Indeed, the event guarantees that $\mathcal{A}$ inputs m^* to the RO and $\mathcal{B}_1$ does not give up $\mathcal{A}$ does not input $\mathsf{initSec}^*$ with $\mathsf{commitSec}^*$ to the RO.

Therefore, we have

$$\begin{aligned}\mathrm{Adv}^{\mathsf{mmIND\text{-}RCCA}}_{\mathsf{mmPKE},1,q_n}(\mathcal{B}_1) &\\ &\geq \Pr(\mathsf{JoinHashed}_{\mathsf{epid}^*})\\ &\geq 1/q_e \Pr(\exists\mathsf{epid} : \mathsf{JoinHashed}_{\mathsf{epid}}).\end{aligned}$$

□

Lemma H.7. *There exists a reduction $\mathcal{B}_2$ such that*

$$\Pr(\exists\mathsf{epid} : \mathsf{CommHashed}_{\mathsf{epid}}) \leq q_e \cdot \mathrm{Adv}^{\mathsf{mmIND\text{-}RCCA}}_{\mathsf{mmPKE},q_e\log(q_n),q_n}(\mathcal{B}_2).$$

[10] Intuitively, the only dependency between the commit and joiner secrets comes from encryptions, so inputting the secrets to the RO requires breaking IND-RCCA.

Proof. We start by describing the reduction $\mathcal{B}$. Recall that SAIK generates mmPKE key pairs and ciphertexts when epochs are created: When a party id_s creates an epoch, it generates a hash chain of secrets, consisting of $\log(q_n)$ path secrets and the commit secret. Each path secret is then hashed to obtain randomness used to generate a single key pair. Moreover, if a new member is added, its new mmPKE key pair is generated by the AKS. Then, id_s sends out all new public keys and a single ciphertext encrypting secrets to different recipients.

The reduction $\mathcal{B}$ runs $\mathcal{A}$, emulating the functionality and the simulator executing SAIK as in hybrid 2 with the following differences. First $\mathcal{B}$ embeds public keys from the mmOW-RCCA game as public keys sent when epochs are created. It generates all secrets itself independently of the key pairs. Further, it picks a random epoch epid^* and a random index $i^* \in [\log(q_n)]$. When epid^* is created, $\mathcal{B}$ asks the challenger for an encryption C^* of the secrets $\mathcal{B}$ generated, but with the i^*-th secret replaced by the challenge message s^* $\mathcal{B}$ is supposed to compute. C^* is then embedded in the sent message. For the the unknown value of the i^*-th secret, $\mathcal{B}_{\mathsf{epid}}$ uses a special symbol *'test'* (it is used for bookkeeping, e.g. to consistently program the RO).[11]

When a party is corrupted, $\mathcal{B}$ corrupts all receivers whose secret keys are in the party's state. When $\mathcal{A}$ sends a new value to the RO, $\mathcal{B}$ checks if it contains its solution s^* and, if so, sends it to the challenger and halts. Otherwise, $\mathcal{B}$ programs the RO consistently with already generated values. Importantly, if the output is key-generation randomness for an mmOW-RCCA receiver, $\mathcal{B}$ corrupts this receiver to obtain it. (Here we use programmability to deal with adaptive corruptions.)

When a party id_r receives a message, $\mathcal{B}$ runs id_r's protocol with the help of the Dec oracle. Note that Dec may output *'test'*, which $\mathcal{B}$ uses for the unknown value of the i^*-th secret.

Precise description of $\mathcal{B}$. At the beginning, $\mathcal{B}$ guesses an epoch $\mathsf{epid}^* \in [q_e]$ and an index $i^* \in [\log(a_n)]$. Then, it runs $\mathcal{A}$, emulating for it the functionality and the simulator by running their code with the following differences.

Recall that the simulator stores a single ratchet tree per epoch. $\mathcal{B}$ modifies these trees by assigning to each node two additional labels: one storing a receiver in the mmOW-RCCA game and one storing a secret. The root's secret stores the epoch's commit secret. The secret of any other internal node stores the path secret from which its key pair was derived. The leaf's secrets are not used. Alternatively, a secret can be set to $\perp$ in case of injections or *'test'* to denote the unknown mmOW-RCCA challenge s^*. A joiner secret can also take value *'test'*. Secret keys in the ratchet tree will not be used.

To emulate the RO, $\mathcal{B}$ keeps a table of programmed input-output pairs. Some inputs and outputs may contain a special symbol *'test'*. The symbol is not in the RO input domain, so it cannot be inputted by $\mathcal{A}$ (but it will be used by $\mathcal{B}$ when the protocol evaluates hashes). Whenever $\mathcal{A}$ sends a new input, $\mathcal{B}$ first checks if it contains its solution and halts if this is the case. Else, it checks if the output should be equal to key-generation randomness derived from a path secret in some ratchet tree node. If so, $\mathcal{B}$ corrupts the node's receiver to obtain the RO output. Else, it programs a fresh value.

Further, $\mathcal{B}$ makes the following changes to how the functionality and simulator process different inputs of $\mathcal{A}$.

- id_s sends. $\mathcal{B}$ generates the new epoch and the message handed to $\mathcal{A}$ as follows:
  (1) Generate the new epoch's path secrets, as well as all secrets in the key schedule at random. If the created epoch is epid^*, replace the i^*-th secret (a path, commit or joiner secret) by *'test'*. Program the RO according to how the secrets are derived.
  (2) Generate the new epoch's ratchet tree: Copy the ratchet tree from id_s's epoch, apply the action and (re-)assign node labels as follows: For each node on id_s's path and, in case of an add, the node of the new member, set the mmOW-RCCA receiver to the next receiver not appearing in any ratchet tree, and set the public key to the public key of its receivers. The path secret of id_s leaf is $\perp$ (since its key pair is generated using fresh randomness) and the path secret of each node above it is set to the secret chosen in Step 1.
  (3) Generate the ciphertext included in the sent packet: If the created epoch is not epid^*, then simply encrypt the secrets. Else, compute the public key vector $\vec{\mathsf{ek}}$ and message vector $\vec{m}$ with the secrets as in the protocol. Let S be the set of all i such that $\vec{m}[i]$ = *'test'*. $\mathcal{B}$ sends $\vec{\mathsf{ek}}$, $\vec{m}$ and S to the challenger to obtain the sent ciphertext.
  (4) Use the above values to complete emulating the functionality and the simulator as in their code.
- id_r receives a message removing it. If the message removes id_r, then it carries no secrets, so $\mathcal{B}$ simply runs id_r's protocol.
- A current member id_r receives a message not removing it. $\mathcal{B}$ first decrypts the path secret s from the packet. Say id_r uses the keys in a ratchet-tree node v to decrypt. If v has an mmOW-RCCA receiver assigned, $\mathcal{B}$ sets s to the output of the decryption oracle. Else, if v has no receiver but it has a path secret, $\mathcal{B}$ derives v's key pair by hashing the secret, programming the RO if necessary, and decrypts s. Else, it rejects the packet on behalf of the simulator.

  After decrypting, $\mathcal{B}$ checks if s is the solution s^* and halts if this is the case. If not, it proceeds as follows.

  $\mathcal{B}$ computes the epoch secret epSec that identifies the epoch into which id_r transitions. The value of epSec is derived from s the same way as in the protocol, where hashes are evaluated using the RO table and the RO is programmed to a fresh value if necessary. Note that some evaluations may involve the symbol *'test'*.

  If id transitions to an injected epoch, $\mathcal{B}$ creates or updates the epoch as follows:
  (1) If the epoch does not exist, create its ratchet tree by applying the action specified in the packet to the ratchet tree from id's current epoch and set public keys, secrets and

[11] One may expect that if $\mathsf{CommHashed}_{\mathsf{epid}}$ occurs, then the challenge can be embedded in the commit secret inputted by $\mathcal{A}$ to the RO. Intuitively, this cannot work, because confidentiality of the commit secret clearly relies on the confidentiality of path secrets before it and of path secrets from which encryption keys were derived.

mmOW-RCCA receivers of all nodes on the re-keyed path to $\perp$. Set the init, epoch and joiner secrets to those derived from epSec.

(2) Let u be the least common ancestor of the sender's and id's leaves in the ratchet tree. Use the decrypted secret s to derive and assign the path secrets and public keys for u and each node above it by evaluating the RO, programming if necessary. (In case the tree already existed, this potentially adds missing secrets to it.)

(3) Assign to each node below u the public key from the packet.

Finally, $\mathcal{B}$ verifies if id_r accepts the packet, as in the simulator. If it does, then $\mathcal{B}$ transitions id_r. Else, it undoes all changes.

- A NEW MEMBER id_r RECEIVES A MESSAGE. In this case id_r receives two ciphertexts, one with its path secret and one with the joiner secret. $\mathcal{B}$ decrypts these secrets as in case a current member receives a message. If one of them is the solution s^*, $\mathcal{B}$ sends it to the challenger and halts.

  Then, $\mathcal{B}$ computes the epoch secret of the epoch into which id_r transitions by hashing the decrypted joiner secret. If this epoch is injected, $\mathcal{B}$ creates or updates it the same way as when current member receives. Note that if the epoch does not exist, $\mathcal{B}$ uses the public part of the ratchet tree from id_r's packet.
- EXPOSE. When id is exposed, $\mathcal{B}$ computes its mmPKE secret keys by hashing the path secrets from the ratchet tree in id's current epoch. $\mathcal{B}$ corrupts the mmOW-RCCA receivers if necessary.

The reduction wins. Assume $\mathsf{CommHashed}_{\mathsf{epid}}$ occurs. We show that there exist epid^* and i^* such that $\mathcal{B}$ wins. We start with a simple observation.

LEMMA H.8. *If* `*all-ind-secs-secure`(epid) *is true, then for each v in τ, v.ek is generated during an honest send.*

PROOF. Take any v in τ. Let epid_0 be the epoch which introduces v.ek and let id_s be its (alleged) creator. Assume towards a contradiction that epid_0 is injected. If $\mathsf{epid}_0 = \mathsf{epid}$, then we immediately get a contradiction with $\mathsf{CommHashed}_{\mathsf{epid}}$. Else, this means that `*ind-secs-bad`($\mathsf{epid}_0, \mathsf{id}_s$) is true. Moreover, no epoch between epid_0 and epid, including epid, is created by id_s or removes it, since this would replace v's keys. Therefore, `*ind-secs-secure`(epid, id_s) is false and `*all-ind-secs-secure`(epid) is false, which contradicts with $\mathsf{CommHashed}_{\mathsf{epid}}$ being true. □

Let τ be the ratchet tree in epid. By Theorem H.8, we can assign to each internal node v in τ a secret: each non-root node is assigned the path secret s encrypted by $\mathcal{B}$ when v's public key was introduced and the root is assigned the commit secret of epid. $\mathsf{CommHashed}_{\mathsf{epid}}$ guarantees that $\mathcal{A}$ inputs to the RO the secret of at least one node, namely the root. Let v^* be a node in τ with the maximal distance from the root whose secret s^* is inputted by $\mathcal{A}$ to the RO. Let epid^* be the epoch before epid which creates v^*'s secret s^*. We claim that $\mathcal{B}$ wins with the guess epid^* and i^* set to v^*'s index.

Indeed, epid^* is honestly created (by Theorem H.8), so $\mathcal{B}$ can embed the challenge. It is left to show that each public key used to encrypt s^* belongs to an uncorrupted mmOW-RCCA receiver. For this, observe that each such key belongs to a node v in v^*'s sub-tree in the ratchet tree τ^* of epid^*. Moreover, v's key does not change between epid^* and epid, since this would replace v^*'s keys as well. By Theorem H.8, this means that v's key belongs to some mmOW-RCCA receiver.

It remains to show that this receiver is not corrupted. This can happen in two cases: 1) if $\mathcal{A}$ inputs to the RO the path secret from which v's key pair was derived or 2) $\mathcal{A}$ corrupts a party holding v's secret key. Case 1) cannot occur for the following reason: v's key pair can only be derived from the secret of an internal node u below v in τ^*. Note that u is also below v^* in τ^*. Therefore, u's secret (and keys) do not change between epid^* and epid, since this would replace v^*'s keys as well. Since v^* has the maximal distance among nodes with secrets inputted to the RO, $\mathcal{A}$ does not input u's secret. Finally, we show that case 2) cannot occur as well.

LEMMA H.9. *If* `*all-ind-secs-secure`(epid) *is true, then for each v in τ, no party holding v.dk is corrupted.*

PROOF. Take any v in τ. Let epid_0 be the epoch which introduces v.ek and let $\mathsf{epid}[1], \ldots, \mathsf{epid}_\ell$ be the epochs after epid_0 that can be reached from it without v's keys being replaced. Note that these epochs form a tree rooted at epid_0.

We first observe that v's subtree is the same in the ratchet trees of all epochs $\mathsf{epid}_0, \ldots, \mathsf{epid}_\ell$, because any modification replaces v's keys. Moreover, epid is one of these epochs, so this subtree is the same as in τ. Let $\mathsf{id}[1], \ldots, \mathsf{id}_n$ be the parties in v's subtree in τ.

Second, we observe that if `*all-ind-secs-secure`(epid) is true, then no id_i is corrupted in any epoch epid_j. The reason is that for any id_i, each epid_j is connected to epid, and epid is one of $\mathsf{epid}_0, \ldots, \mathsf{epid}_\ell$, by a sequence of epochs not created by id_i and not removing or adding it. This is because any such operation would replace v's keys. Therefore, if id_i was corrupted in some epid_j, then the predicate `*ind-secs-secure`(epid, id_i) would be false and the predicate `*all-ind-secs-secure`(epid) would be false, which contradicts $\mathsf{CommitHashed}_{\mathsf{epid}}$.

Finally, it is left to show that v.dk is held only by $\mathsf{id}[1], \ldots, \mathsf{id}_n$ in epochs $\mathsf{epid}_0, \ldots, \mathsf{epid}_\ell$. It is easy to see that this is implied by the following statement:

Statement : Assume an $\mathsf{id}^\perp$ in an epoch $\mathsf{epid}^\perp$ stores a secret key for a ratchet tree node $v^\perp$ such that $v^\perp.\mathsf{dk} = v.\mathsf{ek}$ for some v in τ. Then, there is party id_i and a path between epid and $\mathsf{epid}^\perp$ that does not *heal* id_i, i.e., no epoch on the path is created by id_i, removes it or adds it.

We next prove the above statement by induction on the height of $v^\perp$. For the base case where $v^\perp$ is a leaf, observe that $v^\perp$'s keys are not generated from a seed and that $v^\perp$.dk is only stored by $v^\perp$'s owner after it generates it while creating an epoch. So, $v^\perp.\mathsf{dk} = v.\mathsf{dk}$ can only happen if $v^\perp$.dk is generated by an id_i when it creates an epoch epid_0 before epid. Therefore, epid_0 is a common ancestor of $\mathsf{epid}^\perp$ and epid and can be reached from both epochs by a path that does not heal id_i.

Now assume $v^\perp$ is an internal node and the statement holds for any node with smaller height. Let $\mathsf{epid}_0^\perp$ be the epoch before $\mathsf{epid}^\perp$ that introduces $v^\perp$.dk into the state of $\mathsf{id}^\perp$. Further, let epid_0 be the epoch that introduces v.dk into τ.

We have two cases: First, if $\mathsf{epid}_0^\perp$ is not injected, then we must have $\mathsf{epid}_0^\perp = \mathsf{epid}_0$. The reason is that the only non-injected epoch introducing $v.\mathsf{ek}$ is epid_0. Moreover, all parties transitioning to $\mathsf{epid}_0^\perp = \mathsf{epid}_0$ agree on the public ratchet tree, so $v^\perp = v$ and the subtree of $v^\perp = v$ is the same in epid and $\mathsf{epid}^\perp$. Therefore, the statement is obvious in this case.

Second, assume $\mathsf{epid}_0^\perp$ is injected. Let $u^\perp$ be the node in the ratchet tree of $\mathsf{epid}_0^\perp$ used by $\mathsf{id}^\perp$ to decrypt $v^\perp$'s path secret s. For this proof sketch, we assume that there exists a node u such that $u.\mathsf{ek}$ corresponds to $u^\perp.\mathsf{dk}$ and $u.\mathsf{ek}$ was used to encrypt s when epid_0 was created.[12] This means that u is in the subtree of v in epid_0 and, since this tree is the same as in τ, also in the subtree of v in epid. Further, $u^\perp$ is in the subtree of $v^\perp$ in $\mathsf{epid}_0^\perp$ and, since this tree is the same as in $v^\perp$'s subtree in $\mathsf{epid}^\perp$, also in the subtree of $v^\perp$ in $\mathsf{epid}^\perp$. Moreover, $u^\perp$ is strictly below $v^\perp$ and $u^\perp.\mathsf{dk} = u.\mathsf{dk}$, so by induction hypothesis, there is anid_i and a path between epid and $\mathsf{epid}^\perp$ that does not *heal* id_i. □

□

H.4 SAIK Guarantees Authenticity

The fourth and final Hybrid introduces authenticity, which is formalized by restoring the **authentic** predicate. It is the ideal experiment with $\mathcal{F}_{\text{CGKA}}$.

Hybrid 4: $\text{IDEAL}_{\mathcal{F}^4_{\text{CGKA}},\mathcal{S}^4}$. The functionality $\mathcal{F}^4_{\text{CGKA}}$ uses the original **authentic** predicate from $\mathcal{F}_{\text{CGKA}}$. The simulator $\mathcal{S}^4$ is the same as $\mathcal{S}^4$.

In the remainder of this section, we show that if Sig and MAC are unforgeable and if mmPKE is mmOW-RCCA secure, then SAIK guarantees authenticity, that is, hybrids 3 and 4 are indistinguishable. We note that security of mmPKE is needed e.g. to guarantee secrecy of MAC keys.

Game-based perspective. We observe that hybrids 3 and 4 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 **authentic** 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 hybrids 3 and 4 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. The hybrids are identical unless the following event Forges occurs: There exists an epoch epid with two members id_s and id_r s.t. the following condition holds:

Condition $\mathsf{Forges}(\mathsf{epid}, \mathsf{id}_s, \mathsf{id}_r)$**:** **authentic**$(\mathsf{epid}, \mathsf{id}_s)$ is true and $\mathcal{A}$ makes id_r accept a message that either (A) makes id_r transition to a new epoch epid′ with HG[epid′].inj true (epid′ is injected) and HG[epid].sndr = id_s or (B) removes id_r and id_s did not remove id_r.

[12] This is only false if $\mathcal{A}$ manages to re-encrypt a securely encrypted s under a different key. Being able to do so implies breaking security of mmPKE. Formally, the reduction $\mathcal{B}_{\text{epid}}$ in the full proof searches for the solution s^* in both $\mathcal{A}$'s RO queries and injected messages that it decrypts using the Dec oracle or some other known keys. Accordingly, v^* is taken to be the lowest whose secret is not inputted to the RO or re-encrypted and injected.

Note that (A) implies that asserting `*auth-is-preserved` in $\mathcal{F}_{\text{CGKA}}$ fails, and (B) implies that the assertion on input Receive that removes the receiver fails. These are the only places where $\mathcal{F}_{\text{CGKA}}$ uses **authentic**.

Since epochs in detached trees are not authentic, in the remainder of the proof we only consider epochs in the main history-graph tree. For such an epoch epid, **authentic**$(\mathsf{epid}, \mathsf{id}_s)$ is true if either the group secrets in epid or the individual secrets of id_s are secure. Accordingly, we define two sub-events of Forges depending on which secrets are secure:

Event ForgesSym**:** There exists an epoch epid with two members id_s and id_r such that `*grp-secs-secure(epid)` and $\mathsf{Forges}(\mathsf{epid}, \mathsf{id}_s, \mathsf{id}_r)$ are true.

Event ForgesAsym**:** There exists an epoch epid with two members id_s and id_r such that `*ind-secs-secure(epid, ids)` and $\mathsf{Forges}(\mathsf{epid}, \mathsf{id}_s, \mathsf{id}_r)$ are true.

It remains to bound the probability of each ForgesAsym and ForgesSym.

Asymmetric forgery. We next prove the following lemma

Lemma H.10. *There exists a reduction $\mathcal{B}_1$ such that*

$$\Pr[\mathsf{ForgesAsym}] \leq 2q_e \cdot \mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig}}(\mathcal{B}_1).$$

$\mathcal{B}_1$ emulates hybrid 3 for $\mathcal{A}$ and embeds the challenge key vk^* as one of the verification keys honestly generated during the execution. Keys are honestly generated when group members create epochs during send: each send introduces one new key pair for the sender and, in case of an add, one for the added member (in this case, it is generated by the AKS at the moment of send). Therefore, there are at most $2q_e$ key pairs. $\mathcal{B}_1$ chooses the index of the key to replace by vk^* at random. It uses the Sign oracle to sign honestly sent messages that verify with vk^*. If a party holding the corresponding ssk^* is corrupted, it gives up.

We first show that if ForgesAsym occurs, then the receiver id_r (see the definition of ForgesAsym) verifies the injected message with an honestly generated key vk_r. This means that $\mathcal{B}_1$ has a chance of embedding vk^* as vk_r.

Claim 1. If ForgesAsym occurs then the key vk_r used by id_r to verify the injected message is honestly generated.

Proof. Assume ForgesAsym occurs. Notice that vk_r is introduced into id_r's state when it accepts a message from id_s that transitions it into an ancestor epid_0 of epid_s. Observe first that epid_0 is not injected. The reason is that no epoch between epid_0 and epid_s is created by id_s or removes it, since this would remove vk_r from id_r's state. So, if epid_0 was injected, `*ind-secs-bad(epid0, ids)` would be true and `*ind-secs-secure(epids, ids)` would be false, which contradicts ForgesAsym.

This means that id_s created epid_0 during a send operation and at that point generated an honest verification key vk_s for itself. We know (from the proof that SAIK guarantees consistency) that parties in the same epoch agree on the ratchet tree, which contains all verification keys. Therefore, $\mathsf{vk}_r = \mathsf{vk}_s$, so vk_r is honestly generated. □

We next show that no party holding the secret key sk_r corresponding to vk_r used by id_r is corrupted. This means that if

ForgesAsym occurs and $\mathcal{B}_1$ guesses correctly and embeds vk^* as vk_r, then $\mathcal{B}_1$ does not give up when sk^* is corrupted.

Claim 2. If ForgesAsym occurs then no party holding sk_r corresponding to vk_r used by id_r to verify the injected message is corrupted.

Proof. Assume towards a contradiction that ForgesAsym occurs and a party holding sk_r is corrupted in some epoch $\mathsf{epid}^{\perp}$. Let $\mathsf{epid}_0^{\perp}$ be the epoch before $\mathsf{epid}^{\perp}$ which introduces sk_r into its state.

Observe that (honest) parties only store the signing keys that they generate themselves while creating epochs or that the AKS generates for them when they are added. Moreover, such honestly generated keys are not re-computed and the AKS generates a fresh key pair each time a party is added. This means that the corrupted party is id_s. Moreover, if $\mathsf{epid}_0^{\perp}$ is not injected, then it is the epoch epid_0 which introduces vk_r into the state of id_r. On the other hand if $\mathsf{epid}_0^{\perp}$ is injected, then it must add id_s (and not be created by it).

Observe further that $\mathsf{epid}_0^{\perp}$ and $\mathsf{epid}^{\perp}$ are connected by a path of epochs not created by id_s and not removing it, as this would remove sk_r. The epochs epid_0 and epid_s are connected by a path with the same property. Therefore, if $\mathsf{epid}_0^{\perp}$ is not injected, then epid_s can be reached, through $\mathsf{epid}_0^{\perp} = \mathsf{epid}_0$, from $\mathsf{epid}^{\perp}$ where id_s is corrupted via a path with above property. This makes the prdicate `*ind-secs-secure`$(\mathsf{epid}_s, \mathsf{id}_s)$ false, contradicting ForgesAsym. Moreover, it is easy to see that if $\mathsf{epid}_0^{\perp}$ is injected, then the predicate `*exposed-ind-secs-weak`$(\mathsf{epid}_s, \mathsf{id}_s)$ would be true, which again makes `*ind-secs-secure`$(\mathsf{epid}_s, \mathsf{id}_s)$ false. □

By the two claims, with probability at least $\Pr[\text{ForgesAsym}]/(2q_e)$, both ForgesAsym occurs and id_r uses $\mathsf{vk}_r = \mathsf{vk}^*$ (since there are at most $2q_e$ honestly generated keys). It is left to show that if this happens, then $\mathcal{B}_1$ wins.

Claim 3. If ForgesAsym occurs and $\mathsf{vk}_r = \mathsf{vk}^*$, then $\mathcal{B}_1$ wins.

Proof. Assume ForgesAsym occurs and $\mathsf{vk} = \mathsf{vk}^*$. There are two cases: (A) id_r transitions into an injected epoch epid', (B) id_r is removed.

In case (A), id_r checks that $\mathsf{Sig.vrf}(\mathsf{vk}^*, \mathsf{confTag}', \mathsf{sig})$ is true. Notice that $\mathsf{confTag}'$ uniquely identifies epid', because it is derived by hashing epSec which identifies epid by definition (see the proof of SAIK consistency). Since the epoch is injected and not created by an honest party, no honest party signed $\mathsf{confTag}'$. In particular $\mathcal{B}_1$ never had to send $\mathsf{confTag}'$ to the sign oracle. Therefore, $\mathcal{B}_1$ wins with the forgery $(\mathsf{confTag}', \mathsf{sig})$.

For case (B), id_r checks that $\mathsf{Sig.vrf}(\mathsf{vk}^*, (\mathsf{id}_s, \textit{'rem'}\text{-}\mathsf{id}_r, \mathsf{confTag}), \mathsf{sig})$ is true, where $\mathsf{confTag}$ uniquely identifies epid. ForgesAsym guarantees that id_s did not remove id_r in epid, so it did not sign such a triple. Therefore, $\mathcal{B}_1$ wins with $((\mathsf{id}_s, \textit{'rem'}\text{-}\mathsf{id}_r, \mathsf{confTag}), \mathsf{sig})$. □

Symmetric forgery. We next bound the probability of symmetric forgery.

Lemma H.11. *There exist reductions $\mathcal{B}_2$ and $\mathcal{B}_3$ such that*

$$\Pr[\text{ForgesSym}] \leq q_e \cdot \mathrm{Adv}^{\text{EUF-CMA}}_{\text{MAC}}(\mathcal{B}_2) + q_e^3 \log(q_n) \cdot \mathrm{Adv}^{\text{mmOW-RCCA}}_{\text{mmPKE}, q_e \log(q_n), q_n}(\mathcal{B}_3) + 3q_e^2 q_h / 2^{\kappa}.$$

At a high level, recall that ForgesSym occurs for epoch epid when the adversary $\mathcal{A}$ injects to some id_r in epid a message that either (A) makes id_r transition to a new injected epoch or (B) removes id_r although it was not honestly removed. Triggering (A) requires from $\mathcal{A}$ computing the confirmation tag which is the hash output on epid's initSec and the injected epoch's context. Triggering (B) requires forging a MAC under epid's secret membKey.

To bound the probability of each (A) and (B), we first replace epid's initSec and membKey by random values, independent of the rest of the experiment. To make this possible, the adversary first commits to the epoch epid. This results in a security loss of q_e (for guessing the epoch). Now assuming mmPKE is secure, the change cannot be noticed as long as `*grp-secs-secure` in epid (part of the **confidential** predicate) is true. Since ForgesSym requires this to be true at the moment it occurs (and the predicate is monotone), the change does not affect the probability that ForgesSym occurs (for the first time).

Once initSec is random, in the ROM, the probability of (A) is negligible. Once membKey is random, the probability of (B) is negligible, assuming that MAC is unforgeable.

Formally, we first define the new hybrid.

Hybrid 3*: The same as hybrid 3, except at the beginning $\mathcal{A}$ announces an epoch epid and membKey and initSec in epid are random and independent.

Further, the following event is analogous to ForgesSym but in hybrid 3*.

Event ForgesSym* **:** In hybrid 3*, there exist two members id_s and id_r in epid announced by $\mathcal{A}$ s.t. `*grp-secs-secure`(epid) and $\mathsf{Forges}(\mathsf{epid}, \mathsf{id}_s, \mathsf{id}_r)$ are true.

Next, we show that changing to hybrid 3* does not affect the probability of the bad event much.

Claim 4. There exists a reduction $\mathcal{B}_3$ such that

$$\Pr(\text{ForgesSym}^*) - \Pr(\text{ForgesSym}) \leq q_e^3 \log(q_n) \cdot \mathrm{Adv}^{\text{mmOW-RCCA}}_{\text{mmPKE}, q_e, q_e \log(q_n), q_n}(\mathcal{B}_3) + 3q_e^3 q_n / 2^{\kappa}.$$

Proof. First, observe that if $\mathcal{A}$ triggers ForgesSym in hybrid 3 with probability at least ϵ, then $\mathcal{A}'$ who guesses epid at random triggers ForgesSym* but in hybrid 3 (i.e., membKey and initSec are not random) with probability at least ϵ / q_e.

Second, observe that membKey and initSec are derived the same way as the group key — each key is the result of hashing the epoch secret with a different label. Therefore, the proof analogous to the proof of Theorem H.3 shows that the difference between the probability of ForgesSym* in hybrid 3 and ForgesSym* in hybrid 3* is bounded by $q_e^2 \log(q_n) \cdot \mathrm{Adv}^{\text{mmOW-RCCA}}_{\text{mmPKE}, q_e, q_e \log(q_n), q_n}(\mathcal{B}_3) + 3q_e^2 q_n / 2^{\kappa}$. □

Finally, it is left to bound the probability of ForgesSym*. To this end, we define two sub-events:

Event ForgesSym*(A) **:** In hybrid 3*, there exist two members id_s and id_r in epid announced by $\mathcal{A}$ such that the predicate `*grp-secs-secure`(epid) is true and (A) id_r accepts a message that makes it transition to an epoch epid' with $\mathsf{HG}[\mathsf{epid}'].\mathsf{inj}$ true.

Event ForgesSym * **(B) :** In hybrid 3*, there exist two members id_s and id_r in epid announced by $\mathcal{A}$ such that the predicate `*grp-secs-secure(epid)` is true and (B) id_r accepts a message that removes it but id_s did not remove id_r in epid.

We next bound the probability of each sub-event.

Claim 5. In the ROM, we have

$$\Pr(\mathsf{ForgesSym}^*(A)) \le q_d/2^{\kappa},$$

where q_d is the number of delivered messages.

Proof. Recall that epid′ is identified by its unique epoch secret epSec′, computed as the hash of initSec of epid and the context of epid′. Since in hybrid 3* initSec is random and independent of the experiment, so is epSec′. Further, recall that id_r accepts the message only if the attached confirmation tag confTag′ is equal to the hash of epSec′ with appropriate label. This matches with probability at most $1/2^{\kappa}$. The claim follows by the union bound on the number of injection attempts. ◻

Claim 6. There exists a reduction $\mathcal{B}_2$ such that

$$\Pr(\mathsf{ForgesSym}^*(B)) \le \mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{B}_2).$$

Proof. $\mathcal{B}_2$ emulates hybrid 3* for $\mathcal{A}$, except instead of the MAC key `*mem` in epid announced by $\mathcal{A}$, $\mathcal{B}_2$ uses its EUF-CMA Sign and Verify oracles. Since membKey is random and independent in hybrid 3*, $\mathcal{B}_2$ simulates the experiment perfectly.

It is left to show that if ForgesSym * (B) occurs, then $\mathcal{B}_2$ wins. Assume the event occurs. $\mathcal{B}_2$ outputs the forgery consisting of the tag tag_t from the injected packet removing id_r and the message (id_s, '*rem*'-id_r, confTag) where confTag is the confirmation tag in epid. Since id_r accepted the message, MAC verification passes.

Finally, we claim that $\mathcal{B}_2$ did not query this message to the Sign oracle. Observe that this only happens if an honest party in sends out a MAC over (id_s, '*rem*'-id_r, confTag). Since only id_s MAC's its identity and only parties in epid MAC confTag, this only happens if id_s removes id_r in epid. This is a contradiction with ForgesSym (B). ◻

H.5 Left-Balanced Trees

In this section, we formally define q-ary trees used by SAIK to implement ratchet trees.

Definition H.12 (LBT). For $q, n \in \mathbb{N}$ with $q > 1$, the n^{th} *left-balanced q-ary tree (LBT)*, denoted $\mathsf{LBT}_{q,n}$, is defined as follows. $\mathsf{LBT}_{q,1}$ is the tree consisting of one node. For $n > 1$, if $m = \max\{q^p : p \in \mathbb{N} \wedge q^p < n\}$ and $k = \lfloor n/m \rfloor$, then $\mathsf{LBT}_{q,n}$ is the tree whose root has the first k children equal to $\mathsf{LBT}_{q,m}$ and, if $n - mk > 0$, the $(m+1)$-st child equal to $\mathsf{LBT}_{q,n-mk}$.

Definition H.13 (Full LBT). For $q, n \in \mathbb{N}$, $\mathsf{LBT}_{q,n}$ is full if n is a power of q.

Operation of SAIK requires a procedure $\mathsf{addLeaf}(\tau, v)$ which inserts a leaf v into a ratchet tree τ while preserving certain properties of τ. In particular, addLeaf should preserve node indices v.nodeIdx. They are computed as follows: all nodes are numbered left to right — i.e., according to an in-order depth-first traversal of the tree — starting with 0. See Fig. 19 for an example.

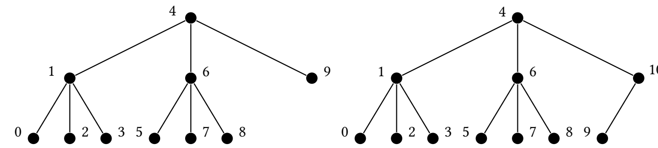


Figure 19: The trees $\mathsf{LBT}_{3,7}$ (left) and $\mathsf{LBT}_{3,8}$ (right) with node indices.

Definition H.14 (addLeaf). The algorithm $\mathsf{addLeaf}(\tau, v)$ takes as input a q-ary tree τ with root r and n nodes, and a fresh leaf v and returns a new tree τ' with v inserted and $v.\mathsf{nodeIdx} = n + 1$.

a) If τ is full, then create a new root r' for τ'. Attach r as the first child of r' and v as the second child.
b) Else if r.children contains only nodes with full subtrees, let $\tau' = \tau$ except v is attached as the next child of r.
c) Else, let u be the first in r.children s.t. its subtree τ_u is not full. Let $\tau' = \tau$ except τ_u is replaced by $\mathsf{addLeaf}(\tau_u, v)$.

The following lemma formalizes the correctness of addLeaf. We prove it in App. H.5.

Theorem H.15. $\tau = \mathsf{LBT}_{q,n} \implies \mathsf{addLeaf}(\tau, v) = \mathsf{LBT}_{q,n+1}$.

Proof. The proof is by strong induction on n. If $n < q$, then the statement easily follows by inspection (only cases a) and b) of addLeaf apply). Fix $n \ge q$ and assume the statement holds for all $k < n$. Let r be the root of τ and let $\text{max-pow}(n) = \max\{q^p + 1 : p \in \mathbb{N} \wedge q^p < n\}$.

If τ is full, then $\text{max-pow}(n+1) = n$. Furthermore, the root of τ' has only two children: $\tau = \mathsf{LBT}_{q,n} = \mathsf{LBT}_{q,\text{max-pow}(n+1)}$ and $\mathsf{LBT}_{q,1}$, so $\tau' = \mathsf{LBT}_{q,n+1}$ per definition.

Else, $\text{max-pow}(n) = \text{max-pow}(n+1)$ (this holds since $n \ge q$). Moreover, it is easy to see that only the last node in r.children can be non-full. This means that the root r' of τ' has the following children (in order):

- All children of the root r of τ which have full subtrees. These subtrees are equal to $\mathsf{LBT}_{q,\text{max-pow}(n)}$ which is the same as $\mathsf{LBT}_{q,\text{max-pow}(n+1)}$.
- If r has no non-full subtrees, then the last child of r' is v with subtree $\mathsf{LBT}_{q,1}$.
- Else if the last child u of r is non-full and equals to $\mathsf{LBT}_{q,x}$ for $x < \text{max-pow}(n)$, then the last child of r' is $\mathsf{LBT}_{q,x+1}$ by induction hypothesis.

Clearly, $\tau' = \mathsf{LBT}_{q,n+1}$ in all cases. ◻