Server-Aided Continuous Group Key Agreement

Joël Alwen[1], Dominik Hartmann[2], Eike Kiltz[2], and Marta Mularczyk[3]

[1] AWS Wickr
alwenjo@amazon.com
[2] Ruhr-University Bochum
dominik.hartmann@rub.de, eike.kiltz@rub.de,
[3] Department of Computer Science, ETH Zurich
mumarta@inf.ethz.ch

Abstract. Continuous Group Key Agreement (CGKA) – or Group Ratcheting – lies at the heart of a new generation of End-to-End (E2E) secure group messaging (SGM) and VoIP protocols supporting very large groups. Yet even for these E2E protocols the primary constraint limiting practical group sizes continues to be their communication complexity. To date, the most important (and only deployed) CGKA is ITK which underpins the IETF's upcoming Messaging Layer Security SGM standard.

In this work, we introduce *server-aided* CGKA (saCGKA) to more precisely model how E2E protocols are usually deployed. saCGKA makes explicit the presence of an (untrusted) server mediating communication between honest parties (as opposed to mere insecure channels of some form or another). Next, we provide a simple and intuitive security model for saCGKA. We modify ITK accordingly to obtain SAIK; a practically efficient and easy to implement saCGKA designed to leverage the server to obtain greatly reduced communication and computational complexity (e.g. relative to ITK). Under the hood, SAIK uses a new type of signature called *Reducible Signature* which we construct from, so called, *Weighted Accumulators*. SAIK obtains further advantages by using *Multi-Recipient Multi-Message* PKE. Finally, we provide empirical data comparing the communication complexity for senders, receivers and the server in ITK vs. three saCGKAs including two instantiations of SAIK.

Table of Contents

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 things 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 need 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 complexities grow linearly in their group sizes. This has imposed natural limitations on real-world group sizes (generally at or below 1000 members).

Consequently, a new generation of E2E protocols are being developed both in academia (e.g. [33, 2, 4, 5, 3, 6, 7]) and industry [16]. 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 achieve this by reducing their communication complexity down to *logarithmic* in the group size; albeit, only under favorable conditions in the execution. This informal property is sometimes termed the *fair-weather complexity* of a protocol.

To date, the most important of these new E2E protocols is the IETF's upcoming SGM standard called the *Messaging Layer Security* (MLS) protocol. It is the product of a collaboration between industry practitioners and academic cryptographers. MLS is in the final stages of standardization and its core components are already seeing initial deployment [32].

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 [2, 39] underlying the Double Ratchet.

Intuitively, a CGKA protocol provides *E2E secure group management* for evolving groups, i.e., groups whose properties such as the set of members, the group name, the set of moderators, etc., can change mid-session. Each change to a group's properties initiates a fresh *epoch*. A CGKA protocol ensures that all group members in an epoch agree on the current group properties as well as on a symmetric *epoch key*, known only to them. The epoch key can then be used to protect application data (e.g. messages or video stream).

MLS too, is (implicitly) based on a CGKA, originally dubbed *TreeKEM* [20]. Since its inception, TreeKEM has undergone several substantial changes [14, 15] before reaching its current form [13, 5]. For clarity, we refer to its current version as *Insider-Secure TreeKEM* (ITK) (using the terminology of [5] where that version was introduced). ITK has already seen its first real world deployment as the backbone of Cisco's Webex conferencing protocol [32].

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 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 [8]. In particular, that work abstracts and generalizes MLS's construction from ITK.

1.1 Our Contributions

The new CGKA in this work is motivated by the observation that in most deployments of E2E protocols the untrusted mailboxing service buffering packets for offline recipients is actually implemented by a full featured (and often highly scalable) server. Thus, we introduce a new (type) of CGKA that leverages the computational capabilities of the server to *greatly* reduce the bandwidth requirements (especially for recipients) over state-of-the art CGKA protocols, and in particular over ITK.

A bit more technically, we introduce a variant of CGKA we call *server-aided* CGKA (saCGKA). Along with the usual protocols for group members, saCGKA also includes a special *Extract* procedure run by the untrusted server to convert a "full packet" uploaded by a sender into an individualized "sub-packet" destined for a particular recipient.[4] We introduce a new, greatly simplified security notion for saCGKA and construct a new practical saCGKA protocol based on ITK called *Server-Aided* ITK (SAIK). To obtain SAIK, we make two key modifications to ITK, sketched out below.

Multi-message multi-recipient PKE. First, we replace ITK's use of standard (CCA secure) PKE with multi-message multi-recipient PKE (mmPKE). As demonstrated by [51], 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, accurately matching the needs of SAIK. It both strengthens and weakens past notions: On the one hand, proving SAIK secure demands that we equip the mmPKE adversary of [51] with adaptive key compromise capabilities. On the other hand, we "only" require *replayable* CCA (RCCA) security [30] rather than full-blown CCA as used in previous' works [4, 5]. This is possible, because our new security model requires agreement on the *semantics* of the transcript of all previous messages, while previous models required agreement on their *values* (which seems overly strict).

We prove the mmPKE construction of [51] satisfies our new notion based on a form of gap Diffie-Hellman assumption, the same as in [51]. The reduction is tight in that the security loss is independent of the number of parties (i.e. key pairs) in the execution. Moreover, we extend the proof to capture mmPKE constructions based on "nominal groups" [1]. Nominal groups abstract the algebraic structure over bit-strings implicit to the X25519 and X448 scalar multiplication functions and corresponding twisted Edwards curves.[48]. In practical terms, this means our proofs also apply to instantiations of [51] that are based on the X25519 and X448 functions.

Reducible signatures. Second, we replace ITK's use of standard (EUF-CMA secure) signatures with a new type called *Reducible Signature (RS).* Intuitively, an RS allows signing a message vector $\vec{m} = (m_1, \ldots, m_n)$ such that later, anyone, e.g. the untrusted mailboxing service, can compute signatures authenticating sub-vectors (i.e., reductions) of $\vec{m}$. The verifier authenticates both the values in the sub-vector and their original positions in $\vec{m}$.[5]

We show how to build Reducible Signatures from standard EUF-CMA secure signatures and a new type of accumulator called Weighted Accumulators. These can be constructed from various assumptions including RSA, lattice-based and pairing-based assumptions. However, we introduce a more practically efficient construction from a collision resistant hash function.

We believe RS to be of interest in their own right, as they naturally lend themselves to a wider class of applications where reducible messages are delivered via resources with computational capabilities. One example is outsourced storage, where a number of files is uploaded to an untrusted cloud. With RS, a data producer can upload a single signature over all files such that later, a consumer can efficiently verify authenticity of a couple downloaded files. Two other use cases are the E2E protocols used by the Ring service [9] and the Wickr Messaging Protocol [43]. In both cases, signing uploaded encrypted content with an RS would allow each receiver to download just the parts of the header (i.e. the "manifest" in Ring parlance) the recipient needs for decryption.[6]

Simpler security model. To analyze the security of SAIK we introduce a new, greatly simplified, security notion for (sa)CGKA which we view as a contribution in its own right. Indeed, past work on CGKA has struggled to provide security notions for CGKA that are both simple enough to be intuitive yet still meaningfully capture the necessary properties. The notion put forth in this work omits/simplifies various

[4] In a typical ITK packet anywhere between half to almost all of the content is not needed by any given receiver beyond verifying the sender's signature over the packet.

[5] This distinguishes RS from redactable signatures [25].

[6] For a messaging application like Wickr which tends to have short plaintexts, redundant data in the header can make up the majority of data downloaded.

security features of a CGKA as long as they can be easily achieved by known practical extensions of a generic CGKA protocol. Thus we obtain a definition focused on the basic properties of a CGKA with the idea that a protocol satisfying our notion can be easily extended to a "full-fledged" CGKA using known techniques.

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 slightly reduces communication complexity (and halves the number of public key operations) compared to ITK. Meanwhile for senders the communication complexity goes from anywhere between logarithmic and even linear in the group size of ITK down to at most logarithmic and even constant for SAIK. Concretely, in a group with 10K parties a receiver in an ITK session may need to download 1.5MB to transition into a new epoch while the same receiver in SAIK downloads no more than 2.2KB.

Outline of the paper. The remainder of this paper is structured as follows. In Sec. 2, we give a more technical overview of the main construction, suitable for practitioners who don't need all the proof details. In Sec. 3 we cover basic notation. Additional preliminaries can be found in App. A. Secs. 4 to 6 cover mmPKE, weighted accumulators and reducible signatures, respectively. Sec. 7 explains the security model for saCGKA. In Sec. 8, we formally state security of SAIK. Sec. 9 presents extensions of SAIK for other primitives, correctness errors and security predicates. Finally, Sec. 10 contains empirical evaluation and comparison of SAIK to previous constructions. An precise description of the SAIK protocol can be found in App. E.

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 [33]. This was soon followed by the first version of TreeKEM in [53] which evolved to add stronger security [53, 14, 57] and more flexible functionality [15] culminating in its current form ITK of [5] reflected in the most recent draft of the MLS RFC [13].

Reducing the communication complexity of TreeKEM and its descendants is not a new goal. *Tainted TreeKEM* [7] exhibits an alternate complexity profile optimized for a setting where the group is managed by a small set of moderators. Recently, [6] 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 [4] and with an eye on practice [3, 5]. Supporting more concurrency has also been a topic of focus as witnessed by the protocols in [22, 15, 58].

Cryptographic models of CGKA security. Defining CGKA security in a simple yet meaningful way has proven to be a serious challenge. Many existing definitions [7, 3, 8, 22] 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 [7], the adversary is not allowed to modify packets. In [3, 8], new packets can be injected but only when authenticity can be guaranteed despite past corruptions (thereby limiting the guarantees the model can capture about session healing). A good indication that these simplifications can be problematic is described in [5]. Namely, each of the above works proves security assuming their CGKA only uses IND-CPA secure encryption. Yet [5] demonstrates an active attack on TreeKEM (that applies equally to similar CGKAs) which uses honest group members as decryption oracles to clearly violate the intuitive security we'd expect from a CGKA.

An interesting outlier is the work of [24] which does permit 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 computation, 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.

In contrast to the above works, [4] aims to capture a quite realistic setting including a fully active adversary that can even set parties' random coins. In [5] this setting is extended to capture *insider* security. That is, active adversaries that can also 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 (seemingly 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 [21]. 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.

Reducible signatures. The Reducible Signatures introduced in this work are conceptually, relatively similar to Redactable Signatures (see e.g. [25, 52, 38, 41]). The latter allow an untrusted censor to remove parts of the signed message vector, hopefully hiding the removed parts from the verifier. This secrecy property is not a goal of our reducible signatures, hence, our constructions are different. See also Remark 5.

mmPKE. mmPKE was first proposed by Kurosawa [47] though their security model was flawed as pointed out and fixed by Bellare et.al [18, 17]. 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 [51] who show that well-known PKE schemes such as ElGamal[40] and Cramer-Shoup [35] 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 [56, 31, 46] 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 to prove ITK secure (at least against adaptive adversaries).

1.2.1 Comparison with [42] The concurrent work of Hashimoto et al. [42] introduces a new variant of CGKA, which we will refer to as *filtered CGKA* (fCGKA), along with an fCGKA protocol called CmPKE. We compare their results to ours below.

Syntax. Both fCGKAs and saCGKA receivers are provided a personalized packet. In an fCGKA this is achieved by having the sender upload one packet per receiver together with a header downloaded by all receivers. In particular, this syntax already implies that an fCGKA will have linear communication complexity for the sender. In contrast, saCGKA is a generalization of fCGKA. In an saCGKA the server uses an extract procedure specified as part of the protocol to derive personalizes packets for each receiver based on a single packet uploaded by the sender.

Communication complexity. The communication complexity of CmPKE and SAIK are incomparable. For a group of size N, in SAIK the size of packet uploaded by a sender varies from $O(\log N)$ to $O(N)$ depending on the sequence of group operations performed so far.[7]. Moreover, implementors have quite some leeway to guide executions to the $O(\log N)$ range (without imposing any constraints on how higher level applications use SAIK). In contrast, for CmPKE senders communication complexity is always $\Omega(N)$. Nor is this merely a consequence of the syntax. Each receiver necessarily receives a different ciphertext (component) specific to their own decryption key.

For receivers the situation is reversed. While in CmPKE they download a packet of size $O(1)$ in SAIK they download data ranging between $O(1)$ and $O(\log N)$. Thus, in a typical execution, reducing receiver bandwidth from logarithmic to constant comes at the cost of increasing sender bandwidth from logarithmic to linear.

Security model. The security model for fCGKA is essentially that of [5]. In particular, in that work an epoch E is identified by the transcript that leads to E, i.e., the sequence of protocol packets leading from the group's initial state to the group's state in epoch E. To adapt this to the fCGKA syntax [42] identify epochs via the sequence of *packet headers* leading to that epoch (as the headers are the only part of the uploaded packet that is in the view of all receivers).

In our model epochs are identified by the transcript's *semantics*, i.e., the sequence of group operations leading to the current state. Apart from arguably being closer to the desired intuition[8], this formal choice is also crucial to expressing saCGKA security. The generality of a generic saCGKA's extract procedure run by the server means that our security model cannot rely on there being any overlap in transcripts of different receivers, even when they are receiving the personalized result of the same uploaded packet. Besides allowing for a more general security notion, another advantage of identifying epochs this way is that we can rely on the weaker notion of mmIND-RCCA security for the underlying PKE primitives we use instead of the CCA flavoured notions used in [42] (and previous works considering fully active adversaries).

[7] In fact, in certain corner cases, uploaded packets can even have size O(1). For example, the commit packets sent by the user at the right most leaf in a ratchet tree with $N = 2^n + 1$ leaves for any natural number n can have size O(1).

[8] members want to agree on *what* events occurred rather than *how* the events were communicated

On the other hand [42] does include the simplifications to the model of [5] that we make. Thus, their notion more completely captures real-world security goals but therefor is also significantly more complex to define and work with in proofs (just as was the case in [5]). Taming the complexity of CGKA security notions (and the proofs they require) while maintaining relevance to real-world settings is an ongoing challenge in the area. Thus, in our view identifying "sound" simplifications can be a contribution in its own right. By "sound" we intuitively mean that the simplified notion, construction and proof can all be easily extended to more complete versions by directly applying known techniques (namely those in [5]). Consequently, sound simplifications let us more clearly focus on the novelties in new work.

Group representation. Finally, at a technical level CmPKE departs from all previous CGKA's (including SAIK) by internally arranging group members in a depth 1 tree (as opposed to the usual left balanced binary tree). Consequently, the core operation of encrypting a secret to the rest of the group except to oneself requires encrypting the secret to each user individually. To this end [42] introduce a collection of novel post-quantum secure multi-recipient PKE schemes designed to minimize the size of ciphertext components destined for individual recipients. Moreover, they prove that it suffices to authenticate (i.e. sign) just the single component of the ciphertext that contains the actual blinded plaintext. This allows the sender to sign just the header component of a CmPKE packet while avoiding having to sign each personalized packet for the senders. In other words, like SAIK, a CmPKE sender need only compute a single signature.

2 Construction Intuitions

With an eye towards practitioners and to build intuition for the technical contributions we begin with an intuitive description of our constructions.

The ITK *protocol.* One of the most important steps in ITK is when a sender generates a sequence of n key pairs and communicates to each group member all n public keys, as well as secret keys from i to n, where i differs among recipients. (In ITK, several group members act as a single recipient; this is irrelevant for this description.) The key pairs are generated in a way such that for each i there exists a single secret (a short bitstring) that can be used to derive all key pairs from i to n. This means that each recipient needs only to obtain one secret and the public keys from 1 to $i-1$.

mmPKE. We use the following notation: $\vec{m}$ is the vector of n secrets, S is the set if all recipient public keys and S_m for $m \in \vec{m}$ is the subset of S consisting of all recipients who receive m. In ITK, the sender simply encrypts each m to each public key in S_m using standard (CCA secure) PKE. In contrast, SAIK redraws its internal abstraction boundaries viewing encrypting $\vec{m}$ to a partition of S as a single call to an mmPKE. This allows SAIK to use the ElGamal-based mmPKE construction of [51]. Compared to ITK, this cuts both the computational complexity of encrypting $\vec{m}$ and the resulting ciphertext size in half (asymptotically as $|\vec{m}|$ grows).

The mmPKE of [51] is straightforward. Let DEM be a data encapsulation scheme and KDF be a Key Derivation Function.[9] Recall that a (generalized) ElGamal encryption of m to public key g^x requires sampling coins r to obtain ciphertext $(g^r, \mathsf{DEM}(k_m, m))$ where $k_m = \mathsf{KDF}(g^{rx}, g^x)$. 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 $k_i = \mathsf{KDF}(g^{rx_i}, g^{x_i})$.

Optimizing for Short Messages. Normally, when 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 other words, for each $m \in M$ choose a fresh key k'_m for an AEAD and encrypt m with k'_m. Then use the mmPKE of [51] to encrypt k'_m to each public key in S_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. We refer to Figure 2 in Section 4 for the details of the construction. We remark that the security notion of [51] does not permit such an optimization.[10]

Reducible signatures. A packet sent by SAIK consists of a number of fields: the n public keys, the common coins g^r and the individual ciphertexts $\mathsf{DEM}(k, m)$. To optimize bandwidth, in SAIK, the mailboxing service forwards to each receiver only the data it needs: the public keys from 1 to $i-1$, g^r and one

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

[10] The reason is that they require the mmPKE to hide if two recipients get the same message. In contrast, it is permitted by our notion parameterized by larger leakage function. The security proof of SAIK works with the larger leakage too.

ciphertext. However, this leaves us with the following question: how can the receiver verify the packet's authenticity given only a subset of the fields? Reducible Signatures (RS) allow doing just that.

In more detail, using an RS scheme, the sender computes a short signature σ over a vector $\vec{m}$ of fields. It then uploads $\vec{m}$ and σ to the server. Later, the RS allows the server to compute a signature σ' authenticating any sub-vector $\vec{m}'$ of $\vec{m}$ out of a collection of possible sub-vectors chosen by the sender. Crucially for SAIK, the receiver verifies not just the values in the sub-vector, but also *which* field in $\vec{m}$ holds each of the values.

Construction. Our construction of RS uses a standard (EUF-CMA secure) signature scheme and a cryptographic accumulator. Roughly speaking, the latter primitive allows computing a short representation *acc* of a set M and "proofs of inclusion" for elements of M; i.e. a proof π_m attesting to the fact that $m \in M$. A proof π_m is verified given only m, *acc* and π_m. In particular, the rest of M is not needed. Additionally, we build a hash chain over the list of public keys *in reverse order*. Specifically, if a user is provided keys 1 to $i-1$, it can complete the hash chain given the $n-i$-th element of it, i.e. the hash up from the n-th key to the i-th. In our RS construction, a signature σ consists of an accumulator *acc* representing the set of all pairs (i, m_i), where m_i is the value of field i, then end of the hash chain h and a standard signature over *acc* and h. The signature σ' for a receiver i consists of the signed *acc*, h, the $n-i$-th part of the hash chain, as well as $\pi_{(i,m_i)}$. (That way, a receiver can authenticate not just the value m_i but also to which field that value is assigned as well as that the keys provided in a package are actually the correct ones.)

Weighted Accumulators. To instantiate our blackbox design, we must choose a concrete accumulator. Of course, using an RS in SAIK only makes sense if the added bandwidth needed to download σ' in place of a single standard signature (for all of $\vec{m}$ as in ITK) is dominated by the bandwidth savings due to downloading the sub-tuple $\vec{m}'$ instead of $\vec{m}$. Thus, it's critical that we try to minimize the size of both the accumulator and the proofs that fields are contained in it.

At first glance this seems like an easy task as there are several RSA-based and pairings-based accumulators with constant sized proofs [10, 50, 36, 19, 12, 55, 28]. However, security of RSA-based constructions requires a large modulus, which makes them practically inefficient. Pairing-based accumulators have another serious drawback if used as part of an RS in SAIK. Namely, producing a proof of inclusion requires either a trapdoor (which can also be used to break the security of the accumulator) or a large set of parameters linear in $|\vec{m}|$ computed from it. For SAIK, this means that the sender needs to choose the trapdoor and send the public parameters to the untrusted server computing proofs. Moreover, since the sender's secrets may leak any time and SAIK aims to provide Post-Compromise Security (PCS), this has to be done anew for each send.

This motivates our hash-based accumulator design. The starting point is the construction of [26], which builds a Merkle tree with leaves containing the fields (i, m_i). The accumulator *acc* is the hash value at the root and a proof that (i, m_i) is consists of the hash values at each node leading into the path from the leaf of (i, m_i) up to the root. In particular, this can be produced by the server given nothing more than $\vec{m}$. As Merkle trees are balanced binary trees this construction results in an accumulator *acc* consisting of a single hash value and proofs of approximately $\log(|\vec{m}|)$ hash values.

It turns out that for SAIK some fields in $\vec{m}$ are needed by far more recipients than others. To use this for further optimizations, we introduce the notion of a *weighted* accumulator which is additionally provided a weight for each element accumulated. Intuitively, the weight of an element denotes the number of times we expect to produce a proof for the element and the goal is to minimize the expected proof length. In our construction, we use the weights to replace the balanced Merkle tree with a Huffman tree, thereby minimizing the weighted sum of the paths to the root. This yields the average proof length of 2 hashes.

For SAIK the weight assigned to a field (i, m_i) is the number of receivers that will download the field. This means that with our accumulator, the proof size sent by the server is constant when amortized across all receivers of a given packet.

Hedged RS. Finally, we augment the above design of an RS with a symmetric key used by the signer to sign and the verifier to verify. This hedges against compromises of the signing key. Indeed, SAIK trivially provides group members with a stream of constantly refreshed shared symmetric keys. That way, even when an adversary learns a users signing keys, as soon as the group transitions to a secure epoch, the leaked keys become useless to the adversary. We implement the symmetric authentication of an RS by adding a MAC of the accumulator as another component of σ. We refer to Figures 5 for the details of the construction.

3 Preliminaries

Additional preliminaries can be found in App. A. For $m, n \in \mathbb{Z}$, we define $[n] := \{1, 2, \dots, n\}$, $[m : n] := \{m, m+1, \dots, n\}$ and if $[m : n] := \varnothing$ if $n < m$. 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 $Y \mathrel{+}\!\leftarrow y$ and appending an entry z to a list L is written as $Z \mathrel{+\!\!+}\!\leftarrow z$. 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 paths and other collections. Moreover, we use object-oriented notation for trees, described below.

$\tau.\mathsf{root}$	Returns the root.
$\tau.\mathsf{nodes}$	Returns the set of all nodes in the tree.
$v.\mathsf{isroot}$	Returns true iff $v = \tau.\mathsf{root}$.
$v.\mathsf{isleaf}$	Returns true iff v has no children.
$v.\mathsf{parent}$	Returns the parent node of v (or $\perp$ if $v.\mathsf{isroot}$).
$v.\mathsf{children}$	Returns the ordered list of v's children (or $\perp$ if $v.\mathsf{isleaf}$).
$v.\mathsf{nodeIdx}$	Returns the node index of v.
$v.\mathsf{depth}$	Returns the length of the path from v to the root $\tau.\mathsf{root}$. Formally, $\tau.\mathsf{root}.\mathsf{depth} = 0$ and $v.\mathsf{depth} = 1 + v.\mathsf{parent}.\mathsf{depth}$ for $v \neq \tau.\mathsf{root}$.

4 Multi-Message Multi-Recipient Encryption

We recall the syntax of mmPKE from [18]. At a high level, mmPKE is standard encryption that supports batching a number of encryption operations together, in order to improve efficiency.[11]

Definition 1 (mmPKE). *A Multi-Message Multi-Receiver Public Key Encryption scheme* $\mathsf{mmPKE} = (\mathsf{KG}, \mathsf{Enc}, \mathsf{Dec}, \mathsf{Ext})$ *consists of the following four algorithms:*

$\mathsf{KG} \xrightarrow{\$} (\mathsf{ek}, \mathsf{dk})$*: Generates a new public/secret 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 public key in* $\vec{\mathsf{ek}}$.

$\mathsf{Ext}(C, i) \to c_i$*: A deterministic algorithm that takes as input a multi-recipient ciphertext* C *and a position index* i *and 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,* $\perp$.

4.1 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 [51], but there are two important differences: First, [51] does not consider corruptions. Second, instead of CCA, we define the slightly weaker notion of Replayable CCA (RCCA). Roughly, for regular encryption, RCCA [30] 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.

Our security notion, called mmIND-RCCA, is formalized by the game in Fig. 1. The game is similar to typical games formalizing 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 whenever a key in $\vec{\mathsf{ek}}^*$ is generated by the adversary or corrupted, the corresponding messages in $\vec{m}_0^*$ and $\vec{m}_1^*$ must be the same. Moreover, the decryption oracle for receiver i outputs a special symbol *'test'* if the plaintext is receiver i's message in either $\vec{m}_0^*$ or $\vec{m}_1^*$ (this is the standard way to define RCCA).

[11] Our syntax is slightly different than in majority of works on mmPKE, where there is no Ext algorithm and instead Enc outputs a vector of individual ciphertexts. Since Ext is deterministic, the syntaxes are equivalent.

```
Game mmIND-RCCA

$\mathbf{Exp}^{\mathsf{mmIND\text{-}RCCA}}_{\mathsf{mmPKE},N,b}(\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2))$
    for $i \in [N]$ do $(\vec{\mathsf{ek}}_i, \mathsf{dk}_i) \leftarrow \mathsf{KG}()$
    $\mathsf{Corrupted} \leftarrow \varnothing$
    $(\vec{\mathsf{ek}}^*, \vec{m}_0^*, \vec{m}_1^*, st) \leftarrow \mathcal{A}_1^{\mathsf{Dec}_1,\mathsf{Cor}}(\mathsf{ek}_1, \ldots, \mathsf{ek}_N)$
    req $|\vec{m}_0^*| = |\vec{m}_1^*| = |\vec{\mathsf{ek}}^*|$
    $c^* \xleftarrow{\$} \mathsf{Enc}(\vec{\mathsf{ek}}^*, \vec{m}_b^*)$
    $b' \leftarrow \mathcal{A}_2^{\mathsf{Dec}_2,\mathsf{Cor}}(c^*, st)$
    req $\mathsf{leak}(\vec{m}_0) = \mathsf{leak}(\vec{m}_1)$
    req $\forall j : \vec{\mathsf{ek}}^*[j] \in \{\mathsf{ek}_i : i \in [N] \setminus \mathsf{Corrupted}\}$
                                        $\vee\ m_0^*[j] = m_1^*[j]$
    return $b'$

Oracle $\mathbf{Dec}_1(i, c)$
    req $i \in [N]$
    return $\mathsf{Dec}(\vec{\mathsf{dk}}[i], c)$

Oracle $\mathbf{Cor}(i)$
    req $i \in [N]$
    $\mathsf{Corrupted}$ +← $i$
    return $\mathsf{dk}_i$

Oracle $\mathbf{Dec}_2(i, c)$
    req $i \in [N]$
    $m \leftarrow \mathsf{Dec}(\vec{\mathsf{dk}}[i], c)$
    if $\exists j : \vec{\mathsf{ek}}^*[j] = \mathsf{ek}_i$
                $\wedge\ m \in \{\vec{m}_0^*[j], \vec{m}_1^*[j]\}$ then
        return 'test'
    else return $m$
```

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

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

KG
    $x \xleftarrow{\$} \mathbb{Z}_p$
    $\mathsf{ek} \leftarrow g^x, \mathsf{dk} \leftarrow x$
    return $(\mathsf{dk}, \mathsf{ek})$

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

$\mathsf{Ext}(i, C = (c_0, \ldots, c_n))$
    return $(c_0, c_i)$

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

Fig. 2: The mmPKE scheme based on Diffie-Hellman from [51]. The scheme requires a group $\mathbb{G}$ of prime order p, generated by g, a data encapsulation mechanism DEM and a hash function Hash.

Finally, the notion is parameterized by the leakage function $\mathsf{leak}(\vec{m})$ which formalizes information about a vector $\vec{m}$ leaked by the encryption function. In this work, we only use the standard leakage that would result from using regular encryption in parallel, that is, $\mathsf{leak}(\vec{m})$ outputs the length of each element of $\vec{m}$. However, we note that SAIK is still secure if mmPKE leaks more information, such as relations between elements of $\vec{m}$ (formally, one-way RCCA is implied even by mmIND-RCCA with larger leakage; see App. D).

Definition 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*

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

where $\mathrm{Exp}^{\mathsf{mmIND\text{-}RCCA}}_{\mathsf{mmPKE},N,b}$ is described in Fig. 1.

4.2 Construction

Fig. 2 recalls the mmPKE scheme based on Diffie-Hellman from [51], here called DH-mmPKE. 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 this work, we show that DH-mmPKE is mmIND-RCCA secure *with adaptive corruptions*, assuming that DSSDH holds in $\mathbb{G}$ and DEM is IND-RCCA secure. Our reduction is relatively tight. In particular, it is independent of the total number of recipients N, which for typical use cases is orders of magnitude larger than the other parameters. Concretely, for CGKA N can reach tens of thousands for CGKA, while other parameters are typically less than 100.

Remark 1 (Nominal groups.). Some practical applications of Diffie-Hellman, most notably CURVE25519 and CURVE448 [48], implement a Diffie-Hellman operation that is not exponentiation in a prime-order group. Instead, such operations can be formalized as so-called nominal groups [1]. In App. B, we generalize and prove Theorem 1 for nominal groups (the security loss is larger by an additive factor). In particular, this means that DH-mmPKE is secure if instantiated with CURVE25519 and CURVE448.

Theorem 1. *Let $\mathbb{G}$ be a group of prime order p with generator g, let DEM be a data encapsulation mechanism and let $\mathsf{mmPKE} = \mathsf{DH\text{-}mmPKE}[\mathbb{G}, g, p, \mathsf{DEM}, \mathsf{Hash}]$. 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 such that*

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

where Hash is modeled as a random oracle, e is the Euler number, n is the length of the encrypted challenge vector, and q_{d_1}, q_c and q_h are the number of queries to the oracles $\mathbf{Dec}_1$, $\mathbf{Cor}$ and the random oracle, respectively.

We only give a proof intuition, which focuses on a technique for getting tighter security with corruptions. See App. C.1 for a full proof.

Proof (intuition). The proof proceeds in a sequence of game hops. We start with G_0, which is the $\mathsf{mmIND\text{-}RCCA}$ experiment with the bit $b = 0$. Then, in G_1 to G_n we switch the elements of the message vector encrypted by the challenger from elements of $\vec{m}_0^*$ to elements of $\vec{m}_1^*$, one by one. G_n is the $\mathsf{mmIND\text{-}RCCA}$ experiment with $b = 1$.

Consider the first hop G_1 where the vector $(\vec{m}_1^*[1], \vec{m}_0^*[2], \dots, \vec{m}_0^*[n])$ is encrypted by the challenger. To show that G_0 and G_1 are indistinguishable, we define two additional hops: First, $G_{0.1}$ is the same as G_0 except if $\vec{m}_0^*[1] \neq \vec{m}_1^*[1]$, then the DEM key k^* used to encrypt $\vec{m}_0^*[1]$ is random and independent. Second, $G_{0.2}$ is the same as $G_{0.1}$ except if $\vec{m}_0^*[1] \neq \vec{m}_1^*[1]$, then $\vec{m}_1^*[1]$ is encrypted under DEM instead of $\vec{m}_0^*[1]$.

Distinguishing between $G_{0.1}$ and $G_{0.2}$ can be easily reduced to DEM security. Observe that the difference between $G_{0.2}$ and G_1 is the same as between G_0 and $G_{0.1}$: k^* is random in $G_{0.2}$ and real in G_1. Therefore, it is left to show that G_0 and $G_{0.1}$ are indistinguishable.

Let $\mathcal{A}$ be any adversary playing G_0 or $G_{0.1}$. $\mathcal{A}$'s distinguishing advantage is upper-bounded by the probability that it inputs to the RO the pre-image of k^*. We call this event E. We next construct a reduction $\mathcal{B}$ who wins if E and some other independent events occur in the experiment with $\mathcal{A}$ and G_0.

$\mathcal{B}$ is given an DSSDH instance $X = g^x$ and $Y = g^y$ and tries to find $Z = g^{xy}$. It runs $\mathcal{A}$ and embeds Y in multiple keys created by G_0 using self-reducability: some ek_i's are *unknown*, i.e., set to Y^{a_i} for random a_i, and some ek_i's are *known*, i.e., set to g^{a_i} for random a_i. Further, $\mathcal{B}$ embeds X as the component c_0^* of the challenge ciphertext C^*. The decrypt oracle is simulated using the DSSDH oracles in the standard way (see App. C.1).

We claim that if 1) only known keys are corrupted, 2) the public key of the first recipient of C^* is unknown and 3) E occurs, then $\mathcal{B}$ wins. Indeed, 1) ensures that $\mathcal{B}$ can simulate the corrupt oracle. Further, 2) means that the hash pre-mage of k^* is implicitly set to a value containing Z. 3) guarantees that $\mathcal{A}$ inputs this value to the RO, which means that $\mathcal{B}$ wins. To maximize the probability of 1) and 2), $\mathcal{B}$ makes each key known with some fixed probability p. One can verify that for $p = 1/q_c$, the probability of 1) and 2) is highest, namely $1/(e^2 q_c)$. (This technique is inspired by Coron [34].) ■

Remark 2. The work [51] proves that $\mathsf{DH\text{-}mmPKE}$ is tightly $\mathsf{mmIND\text{-}CCA}$ secure without corruptions, under the same assumptions as ours. However, their tight security reduction breaks if corruptions are allowed (technically, the reason is that they make all keys "unknown"; see the proof sketch above). A straightforward fix (making exactly one key "unknown") would result in a security loss linear in N. The security loss of our reduction is independent of N but still larger than in [51] (technically, we choose the optimal number of "unknown" keys).

5 Weighted Accumulators

A cryptographic accumulator, as introduced in [19], allows to generate a short *accumulated* representation *acc* of a set and later compute a short proof that an element is in the set represented by *acc*. Weighted accumulators generalize the above notion, in order to allow for more efficient constructions. In particular, a weighted accumulator allows to generate an accumulated representation of a *weighted* set, i.e., a set of pairs (x, w), where w is an integer weight.Later, it is possible to compute a proof that an element x (without weight) is in the weighted set represented by *acc*.

Introducing weights allows to define new measures of efficiency. For example, our construction in Sec. 5.2 optimizes the weighted sum of sizes of all proofs, where the weight of a proof size is the weight of the element whose membership is proved. We stress that weights are irrelevant in the context of security. Formally, the following definition extends the syntax of accumulators from [12] to incorporate weights, but removes the setup algorithm as it isn't necessary for our constructions.

```
Algorithm Huf-wAcc[Hash]

wEval($\{(x_1, w_1), \ldots, (x_n, w_n)\}$)
  $\tau \leftarrow$ labeled-huffman(
                    $(x_1, \ldots, x_n), (w_1, \ldots, w_n)$)
  $acc \leftarrow \tau$.root.label
  return $(acc, aux = ($
                    $(x_1, \ldots, x_n), (w_1, \ldots, w_n)))$

wProve($acc, x, aux$)
  $\tau \leftarrow$ labeled-huffman($aux$)
  $\pi \leftarrow ()$
  Let $v \in \tau$.leaves s.t. $v$.label = Hash(leaf, $x$).
  while $v \neq \tau$.root do
      $v_p \leftarrow v$.parent, $(v_L, v_R) \leftarrow v_p$.children
      if $v = v_L$ then  $\pi \mathbin{+\!\!+}\!\!\leftarrow$ ('$R$', $v_R$.label)
      else $\pi \mathbin{+\!\!+}\!\!\leftarrow$ ('$L$', $v_L$.label)
      $v \leftarrow v$.parent
  return $\pi$

wVrfy($acc, \pi, x$)
  $\hat{h} \leftarrow$ Hash('$leaf$', $x$)
  for $i = 0$ to len($\pi$) $- 1$ do
      if $\pi[i] = $ ('$L$', $h$) then
          $\hat{h} \leftarrow$ Hash('$int$', $h, \hat{h}$)
      else if $\pi[i] = $ ('$R$', $h$) then
          $\hat{h} \leftarrow$ Hash('$int$', $\hat{h}, h$)
      else return 0
  return $\hat{h} = acc$

labeled-huffman($\vec{x}, \vec{w}$)
  Initialize $\tau \leftarrow$ huffman($\vec{x}, \vec{w}$). Compute the following label for each node $v$ in $\tau$: If $\exists i : v = \tau$.leaves[$i$] then $v$.label = Hash('$leaf$', $\vec{x}[i]$), else $v$.label =
  Hash('$int$', $v$.children[0].label, $v$.children[1].label). Return $\tau$
```

Fig. 3: Weighted cryptographic accumulator Huf-wAcc. The function huffman computes the Huffman tree; see App. A.6. See also Sec. 3 for tree-related notation.

Definition 3 (Weighted Accumulators). *A Weighted Accumulator scheme* wAcc *consists of the following three algorithms:*

wEval$(X) \xrightarrow{\$} (acc, aux)$*: Takes as input a set X of element-weight pairs (x, w), where w is a positive integer, and outputs an accumulated representation acc of X and auxiliary information aux.*

wProve$(acc, x, aux) \xrightarrow{\$} \pi$*: Generates a proof π that $(x, w) \in X$ for some set X with the accumulated representation acc and some integer w.*

wVrfy$(acc, x, \pi) \rightarrow 0 \vee 1$*: Verifies the proof π and outputs 1 for accept or 0.*

Remark 3. Some works define cyrptographic accumulators with more features. For instance, dynamic accumulators [28] allow to modify *acc* so that elements are added to the represented set. See e.g. [10] for an overview. Such features are not required for our application. However, weights can be added to any accumulators if this benefits their application.

5.1 Security

We recall the notion of unforgeability (also called *collision resistant/freeness*) from [37] (and adapt the syntax to account for weights).

Definition 4 (Accumulator Unforgeability). *We define the advantage of an adversary $\mathcal{A}$ against the* unforgeability *of a weighted accumulator* Acc *as*

$$\mathrm{Adv}^{\mathsf{UF}}_{\mathsf{Acc}}(\mathcal{A}) = \Pr\left[\begin{array}{c} \nexists w \in \mathbb{N} : (x, w) \in X \wedge (acc, \_) \leftarrow \mathsf{wEval}(X) \\ \wedge\ \mathsf{Vrfy}(acc, x, \pi) = 1 \end{array} \middle| (X, x, \pi) \leftarrow \mathcal{A} \right].$$

5.2 Efficient Construction from Collision-Resistant Hashing

Our construction combines the accumulator scheme of [26] based on Merkle (hash) trees with the idea of Huffman trees (see App. A.6).

The idea of [26] is to build a binary Merkle tree τ on the accumulated set X. That is, τ has one leaf for each element of X. Moreover, each node of τ has assigned a hash value: the value of a leaf is the hash of its element x, and the value of an internal node is the hash of concatenated values of its children. The accumulated representation of X is the value of the root. Membership of an element x is proven by providing the values of all nodes on the co-path of the path from x's leaf of to the root. In order to optimize the weighted sum of the sizes of all proofs, in our scheme τ is not a balanced tree as in [26], but a Huffman tree with the weights of words set to the weights of the elements of X. The formal description of our scheme wAcc is given in Fig. 3.

Huffman Codes are optimal in the sense that the weighted sum of the codeword lengths is minimal. Codeword length depends on the depth of an element in the tree, which is equivalent to proof size in our construction. Therefore, the weighted accumulator minimizes the weighted sum of proof sizes. For equal weights on all words, a Huffman tree is a balanced binary tree, so our accumulator collapses to a regular Merkle tree in that case.

In App. C.2 we prove the following theorem.

Theorem 2. *Let* $\mathsf{Huf\text{-}wAcc}[\mathsf{Hash}]$ *denote the accumulator from Fig. 3 instantiated with a function* $\mathsf{Hash} : \{0,1\}^* \to \{0,1\}^\kappa$. *For any adversary* $\mathcal{A}$, *there exists an adversary* $\mathcal{B}$ *such that*

$$\mathrm{Adv}^{\mathsf{UF}}_{\mathsf{Huf\text{-}wAcc[Hash]}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{CR}}_{\mathsf{Hash}}(\mathcal{B}).$$

Remark 4. In our description, generating proofs for all elements x requires recomputing the Huffman tree for each invocation of $\mathsf{Huf\text{-}wAcc.wProve}$. We note that in practice, caching the tree is often more efficient.

5.3 Other Constructions

Alternatively to our hash-based construction, cryptographic accumulators, and hence also weighted accumulators, can be built from various assumptions, e.g. pairing groups [10, 50, 36], RSA groups [19, 12, 55, 28] and recently from lattices [49, 59]. We compare them to our solution in Sec. 9.4.

6 Reducible Signatures

Consider a scenario in which a sender uploads a vector of messages $\vec{m}$ on a server and later each of a number of recipients wants to download a sub-vector of $\vec{m}$ and verify its authenticity. One naive solution would be to sign $\vec{m}$ using a regular signature scheme. However, this has a high receiver-communication cost, because receivers have to download the elements of $\vec{m}$ they are not interested in. Another naive solution would be to sign each sub-vector of $\vec{m}$ that someone may be interested in. However, this has a high sender-communication cost. Reducible signatures provide a better solution which minimizes both sender and receiver communication cost. Here, the sender uploads a single signature, which can be later personalized for different recipients by the (untrusted) server.

Hedging. Our application requires that message vectors are authenticated under both an asymmetric key pair of the sender and a symmetric key shared among the sender and all recipients. Therefore, we define a variant called *Hedged Reducible Signatures (HRS)*, which means that signing and verification algorithms take as input both an asymmetric signing/verification key and a symmetric key. Intuitively, security is guaranteed as long as either the symmetric or asymmetric signing key is secure (the other one can be arbitrary). Observe that if the symmetric key is set to a fixed value, then HRS collapses to asymmetric reducible signatures. If instead the asymmetric keys are set to a fixed value, then HRS collapses to reducible MAC's.

The reason for defining a single primitive that combines reducible signatures and reducible MAC's instead of signatures and MAC's separately is allowing more efficient schemes. For example, for our construction using separate signatures and MAC's would double the communication cost.

Reducible signatures. A reducible (asymmetric or symmetric) signature scheme allows to sign a vector of messages $\vec{m}$ such that later anyone, without the secret key, can compute a signature σ' on a sub-vector $\vec{m}'$ of $\vec{m}$. The signature on the reduced vector $\vec{m}'$ is typically larger than the original signature on $\vec{m}$, because it contains hints for the verifier about the missing parts of $\vec{m}$.

The signer controls how the signed message $\vec{m}$ can be reduced by specifying a set of allowed reduction patterns rp. Formally, we define

Definition 5. *A* reduction pattern rp *for message vectors of length* n *is any subset of* $[n]$. *A vector of messages* $\vec{m}$ reduced according to rp, *denoted* $\mathsf{rp}(\vec{m})$, *is the sub-vector of* $\vec{m}$ *consisting of* $\vec{m}[i]$ *for* $i \in \mathsf{rp}$.

When checking a signature σ' on $\vec{m}'$, the verifier also specifies a reduction pattern rp according to which, in their belief, the signed vector $\vec{m}$ was reduced to obtain $\vec{m}'$. (Security will require that if verification passes, then this is indeed true). We assume that the verifier knows rp out-of-band (this will be the case for CGKA). However, if needed, it can be sent together with the signature σ'.

Weights. In order to enable more efficient schemes, we define *weighted* HRS. That is, the signer assigns to each allowed reduction pattern rp an integer weight w. This allows to construct schemes that minimize the weighted sum of signatures σ' on all reduced message vectors. Looking ahead, w will be the number of recipients downloading the sender's message reduced according to rp and the weighted sum will be the total receiver communication cost. We note that weights have no meaning in the context of security. Formally, the signing algorithm will take as input a class of allowed reduction patterns defined as follows.

Definition 6. *A* reduction pattern class RPC *for message vectors of length* n *is a set of pairs* (rp, w), *where* rp *is a reduction pattern for message vectors of length* n *and* $w \in \mathbb{N}$ *is a weight.*

Game AEUF-RCMA, SEUF-RCMA

```
$\mathrm{Exp}^{\mathsf{ATK}}_{\mathsf{HRS}}(\mathcal{A})$, $\mathsf{ATK} \in \{\mathsf{AEUF\text{-}RCMA}, \mathsf{SEUF\text{-}RCMA}\}$
  $Q \leftarrow \varnothing$
  if $\mathsf{ATK} = \mathsf{AEUF\text{-}RCMA}$ then
      $(\mathsf{sk}^*, \mathsf{vk}^*) \xleftarrow{\$} \mathsf{KeyGen}$
      $(\vec{m}^*, \sigma^*, k^*, \mathsf{rp}^*) \xleftarrow{\$} \mathcal{A}^{\mathsf{Sign}_A(\cdot,\cdot)}(\mathsf{vk})$
  else if $\mathsf{ATK} = \mathsf{SEUF\text{-}RCMA}$ then
      $k^* \xleftarrow{\$} \mathcal{K}$
      $(\vec{m}^*, \sigma^*, \mathsf{vk}^*, \mathsf{rp}^*) \xleftarrow{\$} \mathcal{A}^{\mathsf{Sign}_S(\cdot,\cdot,\cdot), \mathsf{Verify}(\cdot,\cdot,\cdot)}()$
  for each $(\vec{m}, \mathsf{RPC}) \in Q$ do
      req $\nexists w : (\mathsf{rp}^*, w) \in \mathsf{RPC} \wedge \vec{m}^* = \mathsf{rp}^*(\vec{m})$
  return $\mathsf{Vrfy}(\mathsf{vk}^*, k^*, m^*, \mathsf{rp}^*, \sigma^*)$

Oracle $\mathsf{Sign}_A(k', \vec{m}, \mathsf{RPC})$
  req $\mathsf{RPC} \in \mathsf{RPCS}_{|\vec{m}|}$
  $Q$ +← $(\vec{m}, \mathsf{RPC})$
  return $\mathsf{Sign}(\mathsf{sk}^*, k', \vec{m}, \mathsf{RPC})$

Oracle $\mathsf{Sign}_S(\mathsf{vk}', \vec{m}, \mathsf{RPC})$
  req $\mathsf{RPC} \in \mathsf{RPCS}_{|\vec{m}|}$
  $Q$ +← $(\vec{m}, \mathsf{RPC})$
  return $\mathsf{Sign}(\mathsf{sk}', k^*, \vec{m}, \mathsf{RPC})$

Oracle $\mathsf{Verify}(\mathsf{vk}', \vec{m}, \sigma, \mathsf{rp})$
  return $\mathsf{Vrfy}(\mathsf{vk}', k^*, \vec{m}, \sigma, \mathsf{rp})$
```

Fig. 4: Asymmetric and symmetric unforgeability games for HRS.

Formal syntax. For message vectors of length n, we will denote by RPCS_n the set of all reduction pattern classes supported by an HRS scheme. This means that the supported reduction patterns are all rp contained in some $\mathsf{RPC} \in \mathsf{RPCS}_n$.

Definition 7 (HRS). *A* Hedged Reducible Signature (HRS) *scheme* HRS *for a collection of reduction pattern classes* RPCS_n *for* $n \in \mathbb{N}$ *consists of the following algorithms:*

$\mathsf{KeyGen} \xrightarrow{\$} (\mathsf{sk}, \mathsf{vk})$*: Generates a new signing/verification key pair.*

$\mathsf{Sign}(\mathsf{sk}, k, \vec{m}, \mathsf{RPC}) \xrightarrow{\$} \sigma$*: On input a signing key* sk*, a symmetric key* $k \in \{0,1\}^\kappa$*, a vector of messages* $\vec{m}$ *and a reduction pattern class* $\mathsf{RPC} \in \mathsf{RPCS}_{|m|}$*, outputs a signature* σ*.*

$\mathsf{Reduce}(\mathsf{vk}, \sigma, \vec{m}, \mathsf{rp}) \xrightarrow{\$} \sigma' \vee \bot$*: On input a verification key* vk*, a signature* σ*, a vector of messages* $\vec{m}$ *and a reduction pattern* rp*, outputs a signature* σ' *authenticating* $\mathsf{rp}(\vec{m})$ *(or* $\bot$ *in case the operation fails).*

$\mathsf{Vrfy}(\mathsf{vk}, k, \vec{m}, \mathsf{rp}, \sigma') \rightarrow 0 \vee 1$*: On input a verification key* vk*, a symmetric key* k*, a vector of messages* $\vec{m}$*, a reduction pattern* rp *and a signature* σ'*, outputs* 1 *for accept or* 0 *for reject.*

Definition 8. *An HRS scheme* HRS *for a collection of reduction pattern classes* RPCS_n *for* $n \in \mathbb{N}$ *is (perfectly) correct if for all* $n \in \mathbb{N}$*,* $k \in \{0,1\}^\kappa$*,* $\mathsf{RPC} \in \mathsf{RPCS}_n$*,* $(\mathsf{rp}, w) \in \mathsf{RPC}$ *and message vectors* $\vec{m}$ *of length* n*, we have*

$$\Pr\left[\mathsf{Vrfy}(\mathsf{vk}, k, \mathsf{rp}(\vec{m}), \mathsf{rp}, \sigma') = 1 \;\middle|\; \begin{array}{c} (\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}() \\ \sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, k, \vec{m}, \mathsf{RPC}) \\ \sigma' \leftarrow \mathsf{Reduce}(\mathsf{vk}, \sigma, \vec{m}, \mathsf{rp}) \end{array}\right] = 1.$$

Remark 5. Reducible signatures should not be confused with redactable signatures (see e.g. [25, 52, 38, 41]). The latter allow to sign a message m such that later a censor can, without the secret key, redact parts of m and compute a valid signature on the result. An important security goal is to hide from the verifier the redacted contents, or even that a redaction took place. In contrast, reducible signatures allow the verifier to *check* which reduction pattern was applied, which is in conflict with the goal of redactable signatures.

6.1 Security

We adapt the usual EUF-CMA security notion for Reducible Signatures. Intuitively, an adversary who knows only one of the symmetric secret and the asymmetric secret key should not be able to generate a valid signature. This is formalized in Definition 9, where we define *Symmetric and Asymmetric Unforgeability against Reducible Chosen Message Attacks*(S/AEUF-RCMA).

Definition 9. *For* $\mathsf{ATK} \in \{\mathsf{AEUF\text{-}RCMA}, \mathsf{SEUF\text{-}RCMA}\}$ *and an* HRS*, we define* $\mathrm{Exp}^{\mathsf{ATK}}_{\mathsf{HRS}}$ *in Fig. 4 and the advantage of an adversary* $\mathcal{A}$ *against the* ATK *security of* HRS *as*

$$\mathrm{Adv}^{\mathsf{ATK}}_{\mathsf{HRS}}(\mathcal{A}) = \Pr\left[\mathrm{Exp}^{\mathsf{ATK}}_{\mathsf{HRS}}(\mathcal{A}) = 1\right].$$

In addition to unforgeability, we also require our Reducible Signature to be *Key Committing*. Intuitively, it should not be possible for an adversary to find different symmetric keys, such that a signature is valid under both keys. We define the security notion in App. A.7, Definition 20 and prove that our construction achieves this notion in App. C.4.

```
Algorithm BGM-HRS[Hash, Acc, Sig, MAC]

KeyGen
  return Sig.kg()

Sign (sk, $k$, $\vec{m}$, ($\ell$, $\vec{w}$))
  $X \leftarrow \{((i, \vec{m}[i]), \vec{w}[i]) \mid i \in [\ell]\}$
  $(acc, aux) \xleftarrow{\$} \mathsf{Acc.Eval}(X)$
  $\hat{h} \leftarrow 0$
  for $t = |\vec{m}|$ to $\ell + 1$ do
      $\hat{h} \leftarrow \mathsf{Hash}(\vec{m}[t], \hat{h})$
  $h \leftarrow \mathsf{Hash}(\ell, acc, \hat{h})$
  $\mathsf{tag} \leftarrow \mathsf{MAC.tag}(k, h)$
  $\mathsf{sig} \leftarrow \mathsf{Sig.sign}(\mathsf{ssk}, (h, \mathsf{tag}))$
  return $(\mathsf{sig}, \mathsf{tag}, acc, aux)$

Reduce (vk, $\vec{m}$, $\sigma$, rp $= (\ell, i, j)$)
  parse $(\mathsf{sig}, \mathsf{tag}, acc, aux) \leftarrow \sigma$
  req $i \in [\ell + 1] \wedge j \in [0 : |\vec{m}| - \ell]$
  if $i \in [\ell]$ then
      $\pi \leftarrow \mathsf{Acc.Prove}(acc, \vec{m}[i], aux)$
  else
      $\pi \leftarrow \perp$
  $\hat{h} \leftarrow 0$
  for $t = |\vec{m}|$ to $\ell + j + 1$ do
      $\hat{h} \leftarrow \mathsf{Hash}(\vec{m}[t], \hat{h})$
  return $(\mathsf{sig}, \mathsf{tag}, acc, \pi, \hat{h})$

Vrfy (vk, $k$, $\vec{m}$, rp $= (\ell, i, j)$, $\sigma'$)
  parse $(\mathsf{sig}, \mathsf{tag}, acc, \pi, \hat{h}) \leftarrow \sigma'$
  if $i \notin [\ell + 1]$ then return 0
  if $i \neq \ell + 1$ then
      req $|\vec{m}| = j + 1$
      req $\mathsf{Acc.Vrfy}(acc, (i, \vec{m}[1]), \pi)$
      $t_0 \leftarrow 2$
  else
      req $|\vec{m}| = j$
      $t_0 \leftarrow 1$
  for $t = |\vec{m}|$ to $t_0$ do $\hat{h} \leftarrow \mathsf{Hash}(\vec{m}[t], \hat{h})$
  $h \leftarrow \mathsf{Hash}(\ell, acc, \hat{h})$
  $v_1 \leftarrow \mathsf{MAC.vrf}(k, h, \mathsf{tag})$
  $v_2 \leftarrow \mathsf{Sig.vrf}(\mathsf{vk}, (h, \mathsf{tag}), \mathsf{sig})$
  return $v_1 \wedge v_2$
```

Fig. 5: Hedged reducible signature scheme HRS from accumulator Acc, hash function H, signature scheme Sig and mac MAC. If the message wasn't reduced, verification consists of recomputing h and verifying the signature.

6.2 Construction

Supported reductions. We observe that the messages sent in the CGKA protocol consists of three chunks of data, from which each user requires a different type of subset. First, there is data that every user needs, which is mainly auxiliary data. Then it includes all new public keys from the senders leaf in the tree to the root. Here, each user only needs the prefix up to its lowest common ancestor with the sender. Lastly, it contains a list of ciphertexts, from which each user can decrypt exactly one. Note that the common part for all users can equivalently be handled by including this fixed message as the first message of the list and requiring prefixes to be non-empty. Since such a reduction pattern seems more versatile, we opt to define our patterns this way.

Definition 10. *We let* $\mathsf{RPCS}_n^{\mathrm{SAIK}} = \{\mathsf{RPC}_{\ell,\vec{w}}^{\mathrm{SAIK}} \mid \ell \in [n], \vec{w} : \vec{w} \in \mathbb{N}^{\ell+1}\}$ *for* $n \in \mathbb{N}$ *with*

$$\mathsf{RPC}_{\ell,\vec{w}}^{\mathrm{SAIK}} := \{([i : \min(i, \ell)] \cup [\ell + 1 : \ell + j], \vec{w}[i]) \mid i \in [\ell + 1], j \in [0 : n - \ell]\}$$

$\mathsf{RPC}_{l,\vec{w}}^{\mathrm{SAIK}}$ *is completely described by* ℓ *and* $\vec{w}$ *(for a fixed* n*), so we will use them as input to all algorithms instead. Similarly, for every* $(\mathsf{rp}, w) \in \mathsf{RPC}_{l,\vec{w}}^{\mathrm{SAIK}}$, rp *is uniquely defined by the integers* ℓ, i, j*, so we use that representation here too.*

Construction. Now we show the construction of the Reducible Signature scheme HRS itself. We adapt the construction of [38] to efficiently support the structure of Definition 10 by using a collision-resistant hash function in addition to a EUF-CMA secure signature, a (weighted) cryptographic accumulator and a MAC. Note that the auxiliary accumulator information isn't included in the signature after the reduction as $\mathsf{RPC}_{l,w}^{\mathrm{SAIK}}$ allows only one singleton element. In App. C.3, we prove the following theorem.

Theorem 3. *Let* HRS = BGM-HRS[Hash, Acc, Sig, MAC] *denote the scheme from Fig. 5 instantiated with a hash* Hash*, a signature scheme* Sig*, a cryptographic accumulator* Acc *and a* MAC*. For any adversary* $\mathcal{A}$*, there exist adversaries* $\mathcal{B}_1$*,* $\mathcal{B}_2$ *and* $\mathcal{B}_3$*, and* $\mathcal{B}_1'$*,* $\mathcal{B}_2'$ *and* $\mathcal{B}_3'$*, all with roughly the same runtime as* $\mathcal{A}$*, s.t.*

$$\mathrm{Adv}_{\mathsf{HRS}}^{\mathsf{AEUF\text{-}RCMA}}(\mathcal{A}) \leq \mathrm{Adv}_{\mathsf{Sig}}^{\mathsf{EUF\text{-}CMA}}(\mathcal{B}_1) + \mathrm{Adv}_{\mathsf{Hash}}^{\mathsf{CR}}(\mathcal{B}_2) + \mathrm{Adv}_{\mathsf{Acc}}^{\mathsf{UF}}(\mathcal{B}_3) \text{ and}$$

$$\mathrm{Adv}_{\mathsf{HRS}}^{\mathsf{SEUF\text{-}RCMA}}(\mathcal{A}) \leq \mathrm{Adv}_{\mathsf{MAC}}^{\mathsf{EUF\text{-}CMA}}(\mathcal{B}_1') + \mathrm{Adv}_{\mathsf{Hash}}^{\mathsf{CR}}(\mathcal{B}_2') + \mathrm{Adv}_{\mathsf{Acc}}^{\mathsf{UF}}(\mathcal{B}_3').$$

7 Server-Aided Continuous Group Key Agreement

In this section, we explain saCGKA protocols and our new security model for them. Our model is based on the model of [4]. However, we make a number of simplifications and adjustments, explained in Sec. 7.3.

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.

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 need the language of *history graphs* introduced in [8]. A history graph is a symbolic representation of group evolution. Epochs are represented as nodes and group modifications are represented as directed edges. For example, when Alice in epoch E decides to add Bob, she creates a epoch E' with an edge from E to E'. The graph also stores information about parties' current epochs, adversary's actions, etc.

In an ideal execution, the history graph would be a chain. However, this is not necessarily true. 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 make parties join fake groups invented in its head by injecting messages that invite them. Epochs in fake groups form what we call *detached trees*. In general, the graph is a 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, that is, any two parties in a given epoch reached it by executing the same sequence of group modifications since the latter one joined.

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

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'. Again, an active adversary may destroy authenticity for certain epochs and parties. 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 to the group setting. In the remainder of this section, we present a single simulation-based definition of saCGKA security, which implies all of the above guarantees (among others).

7.1 Security 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.2 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 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.[12]

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 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

[12] 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.

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 still lead the receiver to an honest epoch.

Adaptive corruptions. Our model allows $\mathcal{A}$ to adaptively decide which parties to corrupt, as long as this does not allow it to trivially distinguish the worlds. Specifically, $\mathcal{A}$ can trivially distinguish if a corruption allows it to compute the real group key in an epoch where $\mathcal{F}_{\text{CGKA}}$ 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.

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

7.2 Details of the CGKA Functionality

In this section, we formally define $\mathcal{F}_{\text{CGKA}}$. The code of $\mathcal{F}_{\text{CGKA}}$ is in Fig. 6.

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. [13] 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.

State. $\mathcal{F}_{\text{CGKA}}$ maintains a history graph represented as an array HG, where $\mathsf{HG}[\mathsf{epid}]$ 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 below ($E.\mathsf{inj}$, $E.\mathsf{exp}$ and $E.\mathsf{chall}$ are related to corruptions). Apart from HG, $\mathcal{F}_{\text{CGKA}}$ stores an array CurEp, where $\mathsf{CurEp}[\mathsf{id}]$ denotes the current epoch of the party id.

$E.\mathsf{par}$	The integer identifier of the parent epoch.
$E.\mathsf{sndr}$	The party who created the epoch by performing a group operation.
$E.\mathsf{act}$	The group modification performed when E was created: either up for update, or $\mathsf{add}\text{-}\mathsf{id}_t$ for adding id_t, or $\mathsf{rem}\text{-}\mathsf{id}_t$ for removing id_t.
$E.\mathsf{mem}$	The set of group members.
$E.\mathsf{key}$	The shared group key.
$E.\mathsf{inj}$	A boolean flag indicating if the epoch is injected.
$E.\mathsf{exp}$	The set of group members exposed (i.e., corrupted) in this epoch.
$E.\mathsf{chall}$	A flag indicating if a random group key has been outputted.

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 $\perp$). 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 $\mathcal{F}_{\text{CGKA}}$ (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 $\perp$, 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 $E.\mathsf{chall}$ set from true to false.

7.3 Our saCGKA Security Model vs Previous Models

In order to make the security notion tractable, we made a number of simplifications compared to the models of [4, 5], listed below.

– *Immediate transition:* In our model, a party performing a group operation immediately transitions to the created epoch. In reality, a party would only send a message and wait for an ACK from the mailboxing service before transitioning. If it receives a different message before ACK, it transitions to

[13] 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), $\mathsf{id}_{\mathsf{creator}}$

```
Initialization // Executed on first input.
  $\mathsf{CurEp}[*], \mathsf{HG}[*] \leftarrow \perp$
  $\mathsf{epCtr} \leftarrow 0$
  $E \leftarrow$ *new-ep; $E.\mathsf{sndr} \leftarrow \mathsf{id}_{\mathsf{creator}}$; $E.\mathsf{mem} \leftarrow \{\mathsf{id}_{\mathsf{creator}}\}$
  $\mathsf{HG}[0] \leftarrow E$
  $\mathsf{CurEp}[\mathsf{id}_{\mathsf{creator}}] \leftarrow 0$

Inputs

Input (Send, act), $\mathsf{act} \in \{\mathsf{up}, \mathsf{add\text{-}id}_t, \mathsf{rem\text{-}id}_t\}$ from id
  // Send inputs to adv. and allow it to reject them.
  Send (Send, id, act) to the simulator and receive $ack$.
  req $ack$
  // Compute the new epoch $E$ created by the action.
  $E \leftarrow$ *new-ep; $E.\mathsf{par} \leftarrow \mathsf{CurEp}[\mathsf{id}]$; $E.\mathsf{sndr} \leftarrow \mathsf{id}$; $E.\mathsf{act} \leftarrow \mathsf{act}$
  $E.\mathsf{mem} \leftarrow$ *members($\mathsf{CurEp}[\mathsf{id}], \mathsf{act}$)
  $\mathsf{epCtr}$++; $\mathsf{HG}[\mathsf{epCtr}] \leftarrow E$
  // Enforce security after possible changes to HG.
  assert *HG-is-consistent $\wedge$ *auth-is-preserved
  // Immediately transition id into the created epoch.
  $\mathsf{CurEp}[\mathsf{id}] \leftarrow \mathsf{epCtr}$
  // 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 $I$.
  $\mathsf{epid} \leftarrow \mathsf{CurEp}[\mathsf{id}]$
  req $\mathsf{epid} \neq \perp$
  if $\mathsf{HG}[\mathsf{epid}].\mathsf{key} = \perp$ then
    // Set the key to a random or arbitrary value, depending on confidential.
    if confidential(epid) then
      $\mathsf{HG}[\mathsf{epid}].\mathsf{key} \xleftarrow{\$} \{0,1\}^{\kappa}$
      $\mathsf{HG}[\mathsf{epid}].\mathsf{chall} \leftarrow$ true
    else
      $\mathsf{HG}[\mathsf{epid}].\mathsf{key} \leftarrow I$
  return $\mathsf{HG}[\mathsf{epid}].\mathsf{key}$

Corruption (Expose, id)
  if $\mathsf{CurEp}[\mathsf{id}] \neq \perp$ then // Record that id's state leaked.
    $\mathsf{HG}[\mathsf{CurEp}[\mathsf{id}]].\mathsf{exp}$ +← id
  // Disallow adaptive corruptions to avoid commitment problem.
  only allowed if $\nexists \mathsf{epid} : \mathsf{HG}[\mathsf{epid}].\mathsf{chall} \wedge \neg$confidential(epid)
```

```
Input (Receive, $c$) from id
  // Send inputs to adv and allow it to reject the packet.
  Send (Receive, id, $c$) to the simulator and receive $ack$.
  req $ack$
  // Ask adv. to interpret the packet.
  Receive from the simulator $(\mathsf{sndr}', \mathsf{act}')$.
  if $\mathsf{act}' = \mathsf{rem\text{-}id}$ then
    // Check that sndr' executed rem-id or injections are possible.
    $\mathsf{honestRem} \leftarrow \exists \mathsf{epid} : \big(\mathsf{HG}[\mathsf{epid}].\mathsf{par} = \mathsf{CurEp}[\mathsf{id}]$
                        $\wedge\ \mathsf{HG}[\mathsf{epid}].\mathsf{sndr} = \mathsf{sndr}' \ \wedge \mathsf{HG}[\mathsf{epid}].\mathsf{act} = \mathsf{rem\text{-}id}\big)$
    assert $\mathsf{honestRem} \vee \neg$authentic($\mathsf{HG}[\mathsf{CurEp}[\mathsf{id}]], \mathsf{sndr}'$)
    $\mathsf{CurEp}[\mathsf{id}] \leftarrow \perp$
    return $(\mathsf{sndr}', \mathsf{act}')$
  // Ask adv. to identify the epoch epid where id transitions. If epid = ⊥, a
  new injected epoch is created.
  Receive from the simulator epid.
  if $\mathsf{epid} = \perp$ then
    $E \leftarrow$ *new-ep
    $E.\mathsf{sndr} \leftarrow \mathsf{sndr}'$; $E.\mathsf{act} \leftarrow \mathsf{act}'$; $E.\mathsf{inj} \leftarrow$ true

    if $\mathsf{CurEp}[\mathsf{id}] \neq \perp$ then // If id is in the group, compute $E$.par and
    $E$.mem according to its epoch.
      $E.\mathsf{par} \leftarrow \mathsf{CurEp}[\mathsf{id}]$
      $E.\mathsf{mem} \leftarrow$ *members($\mathsf{CurEp}[\mathsf{id}], \mathsf{act}'$)

    else // If id joined, $E$ is a detached root with arbitrary member set.
      Receive $E$.mem from the simulator; set $E.\mathsf{par} \leftarrow \perp$.
    $\mathsf{epCtr}$++; $\mathsf{HG}[\mathsf{epCtr}] \leftarrow V$
    $\mathsf{epid} \leftarrow \mathsf{epCtr}$
  assert $\mathsf{HG}[\mathsf{epid}] \neq \perp$
  // If a current group member transitions to a detached root, attach it.
  if $\mathsf{CurEp}[\mathsf{id}] \neq \perp \wedge \mathsf{HG}[\mathsf{epid}].\mathsf{par} = \perp$ then
    $\mathsf{HG}[\mathsf{epid}].\mathsf{par} \leftarrow \mathsf{CurEp}[\mathsf{id}]$
  assert $\mathsf{CurEp}[\mathsf{id}] = \perp \vee \mathsf{HG}[\mathsf{epid}].\mathsf{par} = \mathsf{CurEp}[\mathsf{id}]$
  assert $\mathsf{CurEp}[\mathsf{id}] \neq \perp \vee \mathsf{HG}[\mathsf{epid}].\mathsf{act} = \mathsf{add\text{-}id}$
  // Enforce security after possible changes to HG.
  assert *HG-is-consistent $\wedge$ *auth-is-preserved
  $\mathsf{CurEp}[\mathsf{id}] \leftarrow \mathsf{epid}$
  // Transition id and compute its output.
  if $\mathsf{HG}[\mathsf{epid}].\mathsf{act} = \mathsf{add\text{-}id}$ then
    return $(\mathsf{HG}[\mathsf{epid}].\mathsf{sndr}, \mathsf{HG}[\mathsf{epid}].\mathsf{mem})$
  else return $(\mathsf{HG}[\mathsf{epid}].\mathsf{sndr}, \mathsf{HG}[\mathsf{epid}].\mathsf{act})$
```

```
Helper *new-ep
  return new epoch with $\mathsf{sndr} = \perp$, $\mathsf{par} = \perp$, $\mathsf{act} = \perp$, $\mathsf{mem} = \varnothing$,
  $\mathsf{inj} =$ false, $\mathsf{key} = \perp$, $\mathsf{exp} = \varnothing$, $\mathsf{chall} =$ false.

Helper *members(epid, act)
  $G \leftarrow \mathsf{HG}[\mathsf{epid}].\mathsf{mem}$
  if $\mathsf{act} = \mathsf{add\text{-}id}_t$ then $G$ +← $\mathsf{id}_t$
  else if $\mathsf{act} = \mathsf{rem\text{-}id}_t$ then $G$ -← $\mathsf{id}_t$
  if $\mathsf{act} \neq \mathsf{up} \wedge G = \mathsf{HG}[\mathsf{epid}].\mathsf{mem}$ then return $\perp$
  return $G$
```

```
Helper *HG-is-consistent
  // True if HG is a forest and group membership is consistent.
  return true iff
      a) $\forall \mathsf{id}$ s.t. $\mathsf{CurEp}[\mathsf{id}] \neq \perp$ : $\mathsf{id} \in \mathsf{HG}[\mathsf{CurEp}[\mathsf{id}]].\mathsf{mem}$
      b) HG has no cycles
      c) $\forall \mathsf{epid} \in [\mathsf{epCtr}]$ : $\mathsf{HG}[\mathsf{epid}].\mathsf{mem} \neq \perp$
      d) $\forall \mathsf{epid} \in [\mathsf{epCtr}]$ s.t. $\mathsf{HG}[\mathsf{epid}].\mathsf{par} \neq \perp$ :
            $\mathsf{HG}[\mathsf{epid}].\mathsf{mem} =$ *members($\mathsf{HG}[\mathsf{epid}].\mathsf{par}, \mathsf{HG}[\mathsf{epid}].\mathsf{act}$)

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 $\nexists \mathsf{epid} : 1 \leq \mathsf{epid} \leq \mathsf{epCtr} \wedge \mathsf{HG}[\mathsf{epid}].\mathsf{inj}$
                                  $\wedge$ authentic($\mathsf{HG}[\mathsf{epid}].\mathsf{par}, \mathsf{HG}[\mathsf{epid}].\mathsf{sndr}$)
```

Fig. 6: The ideal CGKA functionality.

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. In any case, all states are cleared upon transition.

– *No PKI for authentication keys:* Our ideal world is oblivious to authentication PKI keys. One consequence of this is that we cannot distinguish certain secure epochs from insecure and we are forced to use weaker security predicates. For example, injecting messages inviting new members is always allowed. Moreover, we give no guarantees for parties joining into detached trees (i.e., fake groups created by the adversary). In reality, in protocols like MLS new members use PKI to verify authenticity of messages inviting them and the states of (even fake) groups into which they join. In particular, MLS's tree signing mechanism achieves the guarantee that any epoch (even in a detached tree) where all members use uncorrupted PKI keys is secure [5].

  Since this is not required by our security definition, our protocol ITK does not sign messages for new members or implement tree signing. However, both can be easily added. (Since messages for new members are completely different than for current members, there is no benefit from using HRS.)
– *Deleting group keys:* $\mathcal{F}_{\text{CGKA}}$ does not keep track of whether parties were corrupted before or after outputting the group keys. This information is necessary to enforce that parties' states contain no information about group keys after outputting them. This property is important for FS of the messaging protocol building on saCGKA but it is also trivially achieved by all protocols we are aware of.
– *No randomness corruptions:* We do not consider attacks where the adversary exposes or modifies the randomness used by the protocol. Considering this would make the model quite complicated. For example, it would require dealing with scenarios where the adversary computes a message in its head, injects it, and then makes a party re-compute it.
– *Simplified syntax:* First, our syntax does not include a command for group creation. Instead, the creator starts as a single group member and adds other parties one by one. In reality, one would implement a more efficient solution where many parties can be added at once. For even better efficiency in cases many operations are executed at once, some protocols like MLS use the so-called propose-commit syntax. Unfortunately, this makes the syntax significantly more complex.
– *No correctness guarantees:* We do not model correctness of saCGKA protocols, 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, because now the mailboxing service's protocol is not part of the model. (But it should be described as part of the protocol.)

Another difference between our model and those of [4, 5] is that in [4, 5] 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 that their messages cannot be used to identify injected epochs. We deal with this in a clean way and allow the simulator to identify epochs. That is, epoch identifiers are arbitrary as long as consistency, authenticity and confidentiality hold. This has the additional advantage that security of TreeKEM-like protocols requires IND-RCCA secure encryption instead of IND-CCA as in [4, 5].

We note that another solution would be to parameterize the saCGKA functionality by a relation on received messages that specifies if they lead to the same epoch (each protocol requires a different relation). One disadvantage of this is introducing an additional protocol-related parameter to the ideal world, which makes it more complicated. Moreover, it seems hard to define such a relation for SAIK without changing the protocol. Indeed, this would require a public value received by each party that uniquely identifies the epoch. SAIK's messages do not have an obvious candidate.

8 Security of SAIK

In this section, we formally state security of SAIK. The details of SAIK can be found in App. E, while an intuition is given in Sec. 2. Knowledge of SAIK is not needed for this section.

Security predicates for SAIK

$\textbf{confidential}(\mathsf{epid}) \iff \neg\texttt{*in-det-tree}(\mathsf{epid}) \land \texttt{*grp-secs-secure}(\mathsf{epid})$

$\textbf{authentic}(\mathsf{epid}, \mathsf{id}) \iff \neg\texttt{*in-det-tree}(\mathsf{epid}) \land \big(\mathsf{epid} = 0 \lor \texttt{*grp-secs-secure}(\mathsf{epid}) \lor \texttt{*ind-secs-secure}(\mathsf{epid}, \mathsf{id})\big)$

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

$\texttt{*grp-secs-secure}(\mathsf{epid} = 0) \iff \mathsf{HG}[\mathsf{epid}].\mathsf{exp} = \varnothing$

$\texttt{*grp-secs-secure}(\mathsf{epid} > 0) \iff \mathsf{HG}[\mathsf{epid}].\mathsf{exp} = \varnothing \land \neg\mathsf{HG}[\mathsf{epid}].\mathsf{inj} \land \big(\texttt{*grp-secs-secure}(\mathsf{HG}[\mathsf{epid}].\mathsf{par})$

$\lor\ \texttt{*all-ind-secs-secure}(\mathsf{epid})\big)$

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

$\texttt{*ind-secs-secure}(\mathsf{epid}, \mathsf{id}) \iff \big(\nexists \mathsf{epid}' : \texttt{*share-ind-secs}(\mathsf{epid}, \mathsf{epid}', \mathsf{id}) \land \texttt{*ind-secs-bad}(\mathsf{epid}', \mathsf{id})\big)$

$\boxed{\land\ \neg\texttt{*exposed-ind-secs-weak}(\mathsf{epid}, \mathsf{id})}$

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

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

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

Fig. 7: Security predicates instantiating $\mathcal{F}_{\text{CGKA}}$ constructed by SAIK.

The predicates **confidential** and **authentic** for SAIK are defined in Fig. 7. We define two versions of the predicates: the stronger one that skips the code in $\boxed{\text{boxes}}$ and the weaker one that includes the whole code. The stronger version is not achieved by SAIK. In Sec. 9.2 we sketch how to modify it to achieve it.[14]

We start by explaining the stronger version, which is simpler. First, both predicates give no guarantees for epochs in detached trees until they are attached. From now on, we do not consider epochs in detached trees. The definition is built around the notion of secrets. Secrets are an abstraction for values stored in the protocol state. There are two types of secrets: group secrets, stored in the state of each party, 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`(epid) decides if the group secrets in epid are secure, i.e., not exposed, and the predicate `*ind-secs-secure`(epid, id) decides if id's individual secrets in epid are secure.

The predicate **confidential**(epid) is equivalent to `*grp-secs-secure`(epid), because the group key is a group secret. Further, the predicate **authentic**(epid, id) is true if either `*grp-secs-secure`(epid) is true or `*ind-secs-secure`(epid, id), because both group and id's secrets are necessary to impersonate id in epid. It remains to determine when group and individual secrets are exposed. For group secrets, `*grp-secs-secure`(epid) is defined recursively. The base case states that the group secrets in the root $\mathsf{epid} = 0$ are secure if and only if no party is corrupted in the root. The intuition is that the first secrets are chosen at random by the group creator, and, for FS, corruptions in the descendants of an epoch do not affect the confidentiality of its group secrets.

The induction step states that the group secrets in an epoch $\mathsf{epid} > 0$ are secure if no party is corrupted in epid, the epoch is not created by an injection and either the group secrets in epid's parent epid_p are secure or all individual secrets in epid are secure. Intuitively, this formalizes the requirement that the adversary can learn the group secrets in only three ways: First, from the state of a party corrupted in epid. Second, by injecting them (some injections are disallowed by the authenticity predicate). Third, by computing them the same way an honest *receiver* transitioning to epid would. The latter can only be done using exposed group secrets of epid_p and the receiver's individual secrets in epid_p. Note that the possible receivers are those parties that are group members in epid and that are not epid's creator (who transitions on sending).

Finally, individual secrets of id in epid are exposed whenever there is some other epoch epid′ where id's secrets are the same as in epid and where id was corrupted or its secrets were injected on its behalf. Secrets of id are the same in two epochs if no epoch change 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 epid are not secure in an additional scenario, formalized by `*exposed-ind-secs-weak`. In this scenario, an id_s first

[14] The modification causes a small efficiency loss, but also makes the protocol and proof more complex by introducing many special cases.

honestly adds id and the environment $\mathcal{A}$ injects a message adding id to some other epoch. Finally, id joins $\mathcal{A}$'s epoch and is corrupted before sending any message. See Sec. 9.2 for an explanation of why SAIK is insecure in this case and how it can be modified to be secure (the explanation does not require knowledge of the details of SAIK).

Theorem. Security of SAIK is formalized by the following theorem, proved in App. F. We note that for the mmPKE scheme, we assume a security property called One-Wayness under Relaxed Chosen-Ciphertext Attack, mmOW-RCCA. The notion is strictly weaker than mmIND-RCCA. See App. D for the definition of mmOW-RCCA and a proof that it is implied by mmIND-RCCA.

Theorem 4. *Let $\mathcal{F}_{\text{CGKA}}$ be the CGKA functionality with predicates* **confidential** *and* **authentic** *defined in Fig. 7. Let* SAIK *be instantiated with schemes* mmPKE*,* HRS *and the* HKDF *functions modelled as a hash* Hash*. Let $\mathcal{A}$ be any environment. Denote the output of $\mathcal{A}$ from the real-world execution with* SAIK *and the hybrid setup functionality $\mathcal{F}_{\text{AKS}}$ from Fig. 14 as $\text{REAL}_{\mathsf{SAIK},\mathcal{F}_{\text{AKS}}}(\mathcal{A})$. Further, we denote the output of $\mathcal{A}$ from an ideal-world 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 $\mathcal{B}_5$ such that*

$$\begin{aligned}\Pr\left[\text{IDEAL}_{\mathcal{F}_{\text{CGKA}},\mathcal{S}}(\mathcal{A})=1\right]-\Pr\left[\text{REAL}_{\mathsf{SAIK},\mathcal{F}_{\text{AKS}}}(\mathcal{A})=1\right] &\leq \text{Adv}^{\mathsf{CR}}_{\mathsf{Hash}}(\mathcal{B}_1)\\ &+ q_e^2(q_e+1)\log(q_n)\cdot\text{Adv}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE},q_e\log(q_n),q_n}(\mathcal{B}_2)\\ &+ 3q_h q_e^2(q_e+1)/2^{\kappa}+2q_e\cdot\text{Adv}^{\mathsf{AEUF\text{-}RCMA}}_{\mathsf{HRS}}(\mathcal{B}_3)\\ &+ 2q_e\cdot\text{Adv}^{\mathsf{RKC}}_{\mathsf{HRS}}(\mathcal{B}_4)+q_e\cdot\text{Adv}^{\mathsf{SEUF\text{-}RCMA}}_{\mathsf{HRS}}(\mathcal{B}_5),\end{aligned}$$

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.*

9 Extensions

9.1 Trading-off Server Computation for Communication

Our new (sa)CGKA SAIK assumes that the mailboxing service is a full featured server capable of performing the individualization of messages responsible for the improved bandwidth of our protocol. This individualization mainly consists of generating proofs of accumulation. We assume strong accumulators which allow creation of these proofs without secret knowledge. However, we can trade off this server computation for increased sender communication by having the sender pre-compute all proofs and send them along with the rest of the packet. Receiver communication is unaffected by this change.

For the hash-based accumulator described in Sec. 5, an easy optimization is possible. Instead of computing *all* proofs individually, the sender can instead include the whole Huffman tree in its package. The server then only selects the co-paths in the tree relevant for each user, which is similarly complex to selecting the correct message for each user, which is exactly the task of a standard mailboxing service. This approach increases communication by approximately $2N$ hashes, where N is the number of receiving public keys, i.e. linear in the number of group members in the worst case and logarithmic in the best case.

In conclusion, our saCGKA can be transformed into a regular CGKA at the cost of sender communication while avoiding server computation.

Note that the variant where each submessage is signed individually requires less communication (i.e. N signatures instead of $2N$ hashes) than the transformed SAIK variant and also doesn't require server computation. However, computing each signature requires an expensive public key operation, leading to a shorter sender message, but more computation for the sender than required by the server in SAIK.

9.2 Better Security Predicates

In this section, we sketch the reason why SAIK does not achieve the better security predicates and how it can be modified to achieve them. Intuition about SAIK from Sec. 2 is sufficient to understand this section.

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 vk for HRS, both registered earlier by id. In epochs after E, parties use ek to encrypt messages to id (even before id actually joins) and vk to verify messages from id. Now the adversary $\mathcal{A}$ can create a fake epoch E' adding id with the same ek and vk. Then, id joins E' and is corrupted, leaking

dk and sk. 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. 7) 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 $(\mathsf{vk}_s, \mathsf{sk}_s)$. It sends dk_s and sk_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 vk, and one under vk_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 sk_s. Therefore, confidentiality and authenticity in E is not affected.

9.3 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 11. *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* $\forall j \in [n]$

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

Definition 12. *An HRS scheme* HRS *for a collection of reduction pattern classes* RPCS_n *for* $n \in \mathbb{N}$ *is* δ*-correct if for all* $n \in \mathbb{N}$, $\mathsf{RPC} \in \mathsf{RPCS}_n$, $(\mathsf{rp}, w) \in \mathsf{RPC}$ *and message vectors* $\vec{m}$ *of length* n*, we have*

$$\Pr\left[\mathsf{Vrfy}(\mathsf{vk}, k, \mathsf{rp}(\vec{m}), \mathsf{rp}, \sigma') \neq 1 \middle| \begin{array}{c} (\mathsf{sk}, \mathsf{vk}) \leftarrow \mathsf{KeyGen}() \\ k \xleftarrow{\$} \{0,1\}^{\kappa} \sigma \leftarrow \mathsf{Sign}(\mathsf{sk}, k, \vec{m}, \mathsf{RPC}) \\ \sigma' \leftarrow \mathsf{Reduce}(\mathsf{vk}, \sigma, \vec{m}, \mathsf{rp}) \end{array} \right] \leq \delta.$$

9.4 Other Accumulators

We only define so-called *strong* accumulators, i.e. accumulators where generation and proving membership don't require a secret trapdoor. This is necessary for *server-aided* CGKA, as the mailboxing server is modelled as untrusted and therefore can't know the accumulator trapdoor. We note however, that SAIK can also be instantiated with *weak* accumulators at the cost of increased sender communication. Instead of letting the server compute the proofs, the sender has to pick a fresh trapdoor (and therefore also public parameters) each time it sends a message, precompute *all* required proofs and include the proofs as well as the public parameters in its message to the mailboxing server, which then distributes them to the receivers. The result is not a server-aided CGKA.

Choosing a new trapdoor for every message is needed to preserve post-compromise security as unforgebility of accumulators doesn't necessarily hold if the adversary knows the trapdoor.

The pairing-based accumulator of [50] is a special case we want to elaborate further on. It can be used as a weak accumulator, allowing for fast and compact proof generation. However at the cost of limiting the number of accumulated elements and increasing the size of the public parameters linearly in that bound, it can be transformed into a strong accumulator. We plot the latter variant in Fig. 8.

RSA-based accumulators yield constant size accumulator values and proofs, however the modulus has to be chosen at ~ 15360bits to achieve 256bits of security. For this size, it would require groups of over 2^{30} participants in order for our hash-based accumulator to become worse.

Lattice-based accumulators [49, 59] follow the same hash-tree approach as we do but replace the regular hash function by a lattice-based hash function. The advantage of choosing these (less efficient) hash functions lies in their compatibility with non-interactive zero-knowledge proofs, which makes it possible to proof membership in the accumulator in a zero-knowledge fashion. Since we explicitly *don't* want to hide what was accumulated, the increased size of lattice-based hash functions makes them considerably worse for our application.

Lastly, apart from communication, our hash-based accumulator also saves on computation, as all other variants require a linear amount of public key operations in the number of accumulated elements. Our accumulator only requires a linear amount of hashes, which are substantially more efficient than public key operations.

10 Evaluation

We compare the communication complexity (informally; the "bandwidth") of SAIK (using our weighted accumulator), ITK and a variant of ITK called ITKI where sender uploads an individually tailored and signed message per recipient. We also compare the bandwidth of SAIK when instantiated with different accumulators. Plots for sender and receiver bandwidth can be found in Fig. 8.

Recall that, for all 3 protocols, the size of a packet sent (i.e. the number of fields it contains) can vary quite significantly depending on the session history. Consequently, the plots consist of regions rather than lines. However, the protocols have identical behavior in this regard. For a given group size N, history of past CGKA operations H and a next operation O the number of fields in the packets for O is the same for each of the 3 protocols. The only thing that varies is the size of the signatures and the size of the fields. (So for example, if the next packet sent for SAIK is at the bottom of SAIK's indicated size range then same holds for the other protocols; namely their next packet size will also be at the bottom of their respective size ranges.) We remark that determining "average" or "expected" packet sizes requires first fixing many setting dependent aspects of an execution which is outside the scope of this work. Indeed, it is an important topic of future research to better understand which kinds of policies governing when parties initiate CGKA operations lead to more bandwidth efficient executions for realistic categories of executions.

We highlight some interesting features of the plots. Even for senders (where RS confer no bandwidth savings compared to standard signature), SAIK requires between 81% to just 56% of the bandwidth of ITK due SAIK's use of mmPKE. Conversely, ITKI senders need 138% to 192% of ITK's bandwidth, since ITKI includes 1 signature per recipient.

On the receiver side, ITK is *overwhelmingly* worse than SAIK and ITKI. For example, SAIK receivers need between 60% to about .05% of ITK's bandwidth. Meanwhile, SAIK is slightly worse than ITKI due to the proofs of inclusion required by the RS. But this comes at a large cost in ITKI's sender computational complexity. While ITKI requires 1 signature *per reduced message*, the Huffman-based RS of SAIK requires just 1 signature and between N and $\log(N)$ hash evaluations *per packet*. In essence, relative to ITKI, SAIK trades a bandwidth overhead of 0% to 5% for a large reduction in the sender's computational complexity. (The possible number of reduced messages is the same for all protocols and can range between $\log(N)$ and N.)

We recall that for Huffman-based accumulators, the length of a proof for a given field varies depending on both the total number of fields in a packet and the weight assigned to that field. The most frequently downloaded fields have the shortest proofs. Thus the longest proof for a field in a packet can be between $\log(N)$ and $\log\log(N)$ hash values. But the *average* proof length sent by the server (across all proofs for all recipients of a packet) varies between $\log(N)$ and 2 hash values. In contrast, the balanced trees of Merkle-based accumulator means that the average proof length across fields in a packet have between $\log(N)$ and $\log\log(N)$ hash values. The difference in average case complexity explains the somewhat lower best-case cumulative server outgoing bandwidth for Huffman-based SAIK in plot (d).

Plots (e) and (f) compare SAIK when using different accumulators, namely the Huffman-based one, the RSA-based accumulator of [12] and the pairing-based accumulator of [50]. The Huffman-based accumulator is more efficient than both other variants even in the worst case. We refer to Sec. 9.4 for a more detailed discussion on the choice of accumulators.

Acknowledgements

Dominik Hartmann was supported by German Federal Ministry of Education and Research (BMBF) iBlockchain project. Eike Kiltz was supported by the BMBF iBlockchain project, the EU H2020 PROMETHEUS project 780701, DFG SPP 1736 Big Data, and the Deutsche Forschungsgemeinschaft

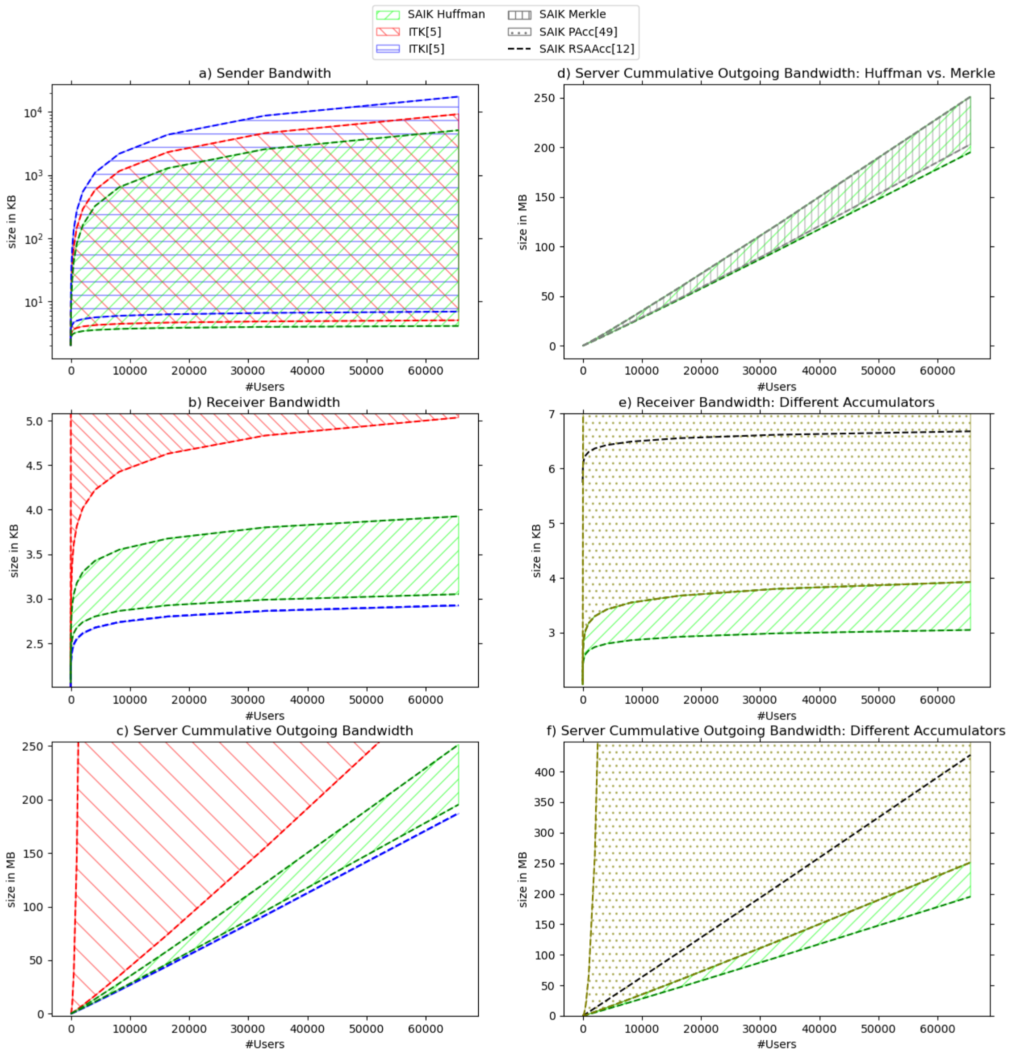


Fig. 8: Bandwidth comparison of SAIK, ITK and ITKI as well as for SAIK instantiated with different accumulators (instantiated with 256-bit security). Lower lines denote the best case while upper lines denote the worst case. Plot (a) shows the sender communication on a log scale. Plot (b) and (e) show individual receiver bandwidth, while (c), (d) and (e) show the cumulative outgoing bandwidth of the server for one protocol message. Plot (d) compares instantiations of SAIK with the Huffman and Merkle-based accumulators. Here, PAcc denotes the pairing-based accumulator of [50] and RSAAcc the RSA-based accumulator form [12]. We elaborate further on the different accumulators in Sec. 9.4.

(DFG, German research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972. Marta Mularczyk was supported by the Zurich Information Security and Privacy Center (ZISC).

References

[1] Joël Alwen, Bruno Blanchet, Eduard Hauck, Eike Kiltz, Benjamin Lipp, and Doreen Riepel. Analysing the HPKE standard. pages 87–116, 2021. Full Version: `https://ia.cr/2020/1499`.

[2] Joël Alwen, Sandro Coretti, and Yevgeniy Dodis. The double ratchet: Security notions, proofs, and modularization for the Signal protocol. In Yuval Ishai and Vincent Rijmen, editors, *EUROCRYPT 2019, Part I*, volume 11476 of *LNCS*, pages 129–158. Springer, Heidelberg, May 2019.

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

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

[5] Joël Alwen, Daniel Jost, and Marta Mularczyk. On the insider security of MLS. Cryptology ePrint Archive, Report 2020/1327, 2020. `https://eprint.iacr.org/2020/1327`.

[6] Joël Alwen, Benedikt Auerbach, Mirza Ahad Baig, Miguel Cueto Noval, Karen Klein, Guillermo Pascual-Perez, Krzysztof Pietrzak, and Michael Walter. Grafting key trees: Efficient key management for overlapping groups. In *Theory of Cryptography — TCC 2021*, 2021. Full version: `https://ia.cr/2021/1158`.

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

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

[9] Amazon. Ring : End-2-end encryption, 2021. `https://bit.ly/3mc9JDF`.

[10] Man Ho Au, Patrick P. Tsang, Willy Susilo, and Yi Mu. Dynamic universal accumulators for DDH groups and their application to attribute-based anonymous credential systems. In Marc Fischlin, editor, *CT-RSA 2009*, volume 5473 of *LNCS*, pages 295–308. Springer, Heidelberg, April 2009.

[11] Michael Backes, Markus Dürmuth, Dennis Hofheinz, and Ralf Küsters. Conditional reactive simulatability. In Dieter Gollmann, Jan Meier, and Andrei Sabelfeld, editors, *ESORICS 2006*, volume 4189 of *LNCS*, pages 424–443. Springer, Heidelberg, September 2006.

[12] Niko Bari and Birgit Pfitzmann. Collision-free accumulators and fail-stop signature schemes without trees. In Walter Fumy, editor, *EUROCRYPT'97*, volume 1233 of *LNCS*, pages 480–494. Springer, Heidelberg, May 1997.

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

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

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

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

[17] M. Bellare, A. Boldyreva, K. Kurosawa, and J. Staddon. Multirecipient encryption schemes: How to save on bandwidth and computation without sacrificing security. *IEEE Transactions on Information Theory*, 53(11):3927–3943, 2007.

[18] Mihir Bellare, Alexandra Boldyreva, and Jessica Staddon. Randomness re-use in multi-recipient encryption schemeas. In Yvo Desmedt, editor, *PKC 2003*, volume 2567 of *LNCS*, pages 85–99. Springer, Heidelberg, January 2003.

[19] Josh Cohen Benaloh and Michael de Mare. One-way accumulators: A decentralized alternative to digital sinatures (extended abstract). In Tor Helleseth, editor, *EUROCRYPT'93*, volume 765 of *LNCS*, pages 274–285. Springer, Heidelberg, May 1994.

[20] Karthikeyan Bhargavan, Richard Barnes, and Eric Rescorla. TreeKEM: Asynchronous Decentralized Key Management for Large Dynamic Groups, May 2018. Published at `https://mailarchive.ietf.org/arch/msg/mls/e3ZKNzPC7Gxrm3Wf0q96dsLZoD8`.

[21] Karthikeyan Bhargavan, Benjamin Beurdouche, and Prasad Naldurg. Formal Models and Verified Protocols for Group Messaging: Attacks and Proofs for IETF MLS. Research report, Inria Paris, December 2019.

[22] Alexander Bienstock, Yevgeniy Dodis, and Paul Rösler. On the price of concurrency in group ratcheting protocols. Cryptology ePrint Archive, Report 2020/1171, 2020. `https://ia.cr/2020/1171`.

[23] Jacqueline Brendel, Cas Cremers, Dennis Jackson, and Mang Zhao. The provable security of Ed25519: Theory and practice. Cryptology ePrint Archive, Report 2020/823, 2020. `https://eprint.iacr.org/2020/823`.

[24] Chris Brzuska, Eric Cornelissen, and Konrad Kohbrok. Cryptographic security of the mls rfc, draft 11. Cryptology ePrint Archive, Report 2021/137, 2021. `https://ia.cr/2021/137`.

[25] Christina Brzuska, Heike Busch, Özgür Dagdelen, Marc Fischlin, Martin Franz, Stefan Katzenbeisser, Mark Manulis, Cristina Onete, Andreas Peter, Bertram Poettering, and Dominique Schröder. Redactable signatures for tree-structured data: Definitions and constructions. In Jianying Zhou and Moti Yung, editors, *ACNS 10*, volume 6123 of *LNCS*, pages 87–104. Springer, Heidelberg, June 2010.

[26] Philippe Camacho, Alejandro Hevia, Marcos A. Kiwi, and Roberto Opazo. Strong accumulators from collision-resistant hashing. In Tzong-Chen Wu, Chin-Laung Lei, Vincent Rijmen, and Der-Tsai Lee, editors, *ISC 2008*, volume 5222 of *LNCS*, pages 471–486. Springer, Heidelberg, September 2008.

[27] Jan Camenisch, Robert R. Enderlein, Stephan Krenn, Ralf Küsters, and Daniel Rausch. Universal composition with responsive environments. In Jung Hee Cheon and Tsuyoshi Takagi, editors, *ASIACRYPT 2016, Part II*, volume 10032 of *LNCS*, pages 807–840. Springer, Heidelberg, December 2016.

[28] Jan Camenisch and Anna Lysyanskaya. Dynamic accumulators and application to efficient revocation of anonymous credentials. In Moti Yung, editor, *CRYPTO 2002*, volume 2442 of *LNCS*, pages 61–76. Springer, Heidelberg, August 2002.

[29] Ran Canetti. Universally composable security: A new paradigm for cryptographic protocols. In *42nd FOCS*, pages 136–145. IEEE Computer Society Press, October 2001.

[30] Ran Canetti, Hugo Krawczyk, and Jesper Buus Nielsen. Relaxing chosen-ciphertext security. In Dan Boneh, editor, *CRYPTO 2003*, volume 2729 of *LNCS*, pages 565–582. Springer, Heidelberg, August 2003.

[31] Haitao Cheng, Xiangxue Li, Haifeng Qian, and Di Yan. CCA secure multi-recipient KEM from LPN. In David Naccache, Shouhuai Xu, Sihan Qing, Pierangela Samarati, Gregory Blanc, Rongxing Lu, Zonghua Zhang, and Ahmed Meddahi, editors, *ICICS 18*, volume 11149 of *LNCS*, pages 513–529. Springer, Heidelberg, October 2018.

[32] Cisco. Zero-trust security for webex, 2021. `https://www.cisco.com/c/en/us/solutions/collateral/collaboration/white-paper-c11-744553.pdf`.

[33] Katriel Cohn-Gordon, Cas Cremers, Luke Garratt, Jon Millican, and Kevin Milner. On ends-to-ends encryption: Asynchronous group messaging with strong security guarantees. In David Lie, Mohammad Mannan, Michael Backes, and XiaoFeng Wang, editors, *ACM CCS 2018*, pages 1802–1819. ACM Press, October 2018.

[34] Jean-Sébastien Coron. On the exact security of full domain hash. In Mihir Bellare, editor, *CRYPTO 2000*, volume 1880 of *LNCS*, pages 229–235. Springer, Heidelberg, August 2000.

[35] Ronald Cramer and Victor Shoup. Universal hash proofs and a paradigm for adaptive chosen ciphertext secure public-key encryption. In Lars R. Knudsen, editor, *EUROCRYPT 2002*, volume 2332 of *LNCS*, pages 45–64. Springer, Heidelberg, April / May 2002.

[36] Ivan Damgård and Nikos Triandopoulos. Supporting non-membership proofs with bilinear-map accumulators. Cryptology ePrint Archive, Report 2008/538, 2008. `https://eprint.iacr.org/2008/538`.

[37] David Derler, Christian Hanser, and Daniel Slamanig. Revisiting cryptographic accumulators, additional properties and relations to other primitives. In Kaisa Nyberg, editor, *CT-RSA 2015*, volume 9048 of *LNCS*, pages 127–144. Springer, Heidelberg, April 2015.

[38] David Derler, Henrich C. Pöhls, Kai Samelin, and Daniel Slamanig. A general framework for redactable signatures and new constructions. In Soonhak Kwon and Aaram Yun, editors, *ICISC 15*, volume 9558 of *LNCS*, pages 3–19. Springer, Heidelberg, November 2016.

[39] Nir Drucker and Shay Gueron. Continuous key agreement with reduced bandwidth. In Shlomi Dolev, Danny Hendler, Sachin Lodha, and Moti Yung, editors, *Cyber Security Cryptography and Machine Learning - Third International Symposium, CSCML 2019, Beer-Sheva, Israel, June 27-28, 2019, Proceedings*, volume 11527 of *Lecture Notes in Computer Science*, pages 33–46. Springer, 2019.

[40] Taher ElGamal. A public key cryptosystem and a signature scheme based on discrete logarithms. In G. R. Blakley and David Chaum, editors, *CRYPTO'84*, volume 196 of *LNCS*, pages 10–18. Springer, Heidelberg, August 1984.

[41] Stuart Haber, William Horne, and Miaomiao Zhang. Efficient transparent redactable signatures with a single signature invocation. Cryptology ePrint Archive, Report 2016/1165, 2016. `https://eprint.iacr.org/2016/1165`.

[42] Keitaro Hashimoto, Shuichi Katsumata, Eamonn Postlethwaite, Thomas Prest, and Bas Westerbaan. A concrete treatment of efficient continuous group key agreement via multi-recipient pkes, 2021. Full Version: `https://ia.cr/2021/1407`.

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

[44] David A Huffman. A method for the construction of minimum-redundancy codes. *Proceedings of the IRE*, 40(9):1098–1101, 1952.

[45] Daniel Jost, Ueli Maurer, and Marta Mularczyk. A unified and composable take on ratcheting. In Dennis Hofheinz and Alon Rosen, editors, *TCC 2019, Part II*, volume 11892 of *LNCS*, pages 180–210. Springer, Heidelberg, December 2019.

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

[47] Kaoru Kurosawa. Multi-recipient public-key encryption with shortened ciphertext. In David Naccache and Pascal Paillier, editors, *PKC 2002*, volume 2274 of *LNCS*, pages 48–63. Springer, Heidelberg, February 2002.

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

[49] Benoît Libert, San Ling, Khoa Nguyen, and Huaxiong Wang. Zero-knowledge arguments for lattice-based accumulators: Logarithmic-size ring signatures and group signatures without trapdoors. In Marc Fischlin and Jean-Sébastien Coron, editors, *EUROCRYPT 2016, Part II*, volume 9666 of *LNCS*, pages 1–31. Springer, Heidelberg, May 2016.

[50] Lan Nguyen. Accumulators from bilinear pairings and applications. In Alfred Menezes, editor, *CT-RSA 2005*, volume 3376 of *LNCS*, pages 275–292. Springer, Heidelberg, February 2005.

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

[52] Henrich Christopher Pöhls and Kai Samelin. On updatable redactable signatures. In Ioana Boureanu, Philippe Owesarski, and Serge Vaudenay, editors, *ACNS 14*, volume 8479 of *LNCS*, pages 457–475. Springer, Heidelberg, June 2014.

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

[54] Phillip Rogaway. Formalizing human ignorance. In Phong Q. Nguyen, editor, *Progress in Cryptology - VIETCRYPT 06*, volume 4341 of *LNCS*, pages 211–228. Springer, Heidelberg, September 2006.

[55] Tomas Sander. Efficient accumulators without trapdoor extended abstracts. In Vijay Varadharajan and Yi Mu, editors, *ICICS 99*, volume 1726 of *LNCS*, pages 252–262. Springer, Heidelberg, November 1999.

[56] Nigel P. Smart. Efficient key encapsulation to multiple parties. In Carlo Blundo and Stelvio Cimato, editors, *SCN 04*, volume 3352 of *LNCS*, pages 208–219. Springer, Heidelberg, September 2005.

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

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

[59] Zuoxia Yu, Man Ho Au, Rupeng Yang, Junzuo Lai, and Qiuliang Xu. Lattice-based universal accumulator with nonmembership arguments. In Willy Susilo and Guomin Yang, editors, *ACISP 18*, volume 10946 of *LNCS*, pages 502–519. Springer, Heidelberg, July 2018.

Supplementary Material

A Additional Preliminaries

A.1 Assumptions

The security of our mmPKE construction, same as that of [51], 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 [51]). We recall it in Definition 13. Intuitively, it states that CDH is hard given access to a DDH-oracle for both CDH inputs.

Definition 13 (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*

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

where $\mathsf{O}_x, \mathsf{O}_y$ 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.

A.2 Universal Composability

We formalize security in the universal composability (UC) framework [29]. We moreover use the modification of responsive environments introduced by Camenisch et al. [27] 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) [4].[15] In a nutshell, this corruption model allows the adversary to repeatedly corrupt parties by sending corruption messages of the form $(\mathtt{Expose}, \mathsf{id})$, 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 [4] (based on prior work by Backes et al. [11] and Jost et al. [45]) 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.3 Collision-Resistant Hashing

We define collision resistance for hash functions in Definition 14.

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

Definition 14 (Collision Resistance). *Let* $\mathsf{Hash} : \{0,1\}^* \to \{0,1\}^\kappa$ *be a hash function. We define the advantage of an adversary* $\mathcal{A}$ *against the collision resistance of* Hash *as*

$$\mathrm{Adv}^{\mathsf{CR}}_{\mathsf{Hash}}(\mathcal{A}) = \Pr\left[\mathsf{Hash}(x_1) = \mathsf{Hash}(x_2) \mid (x_1, x_2) \overset{\$}{\leftarrow} \mathcal{A}\right].$$

Note that for simplicity, we define *unkeyed* hash functions. Generally, these hash functions aren't collision resistant, since there always exists some algorithm that has a hard-coded collision. However since such algorithms are *unknown* for real-world hash functions and we give constructive reductions (i.e. fully black-box reductions where access to an algorithm breaking our building blocks directly yields an algorithm finding a collision), we ignore these existing but unknown algorithms. For a more in-depth discussion, see [54].

A.4 Data Encapsulation Meachanism(DEM)

A DEM is the symmetric equivalent of a PKE scheme. We recall it in Definition 15.

Definition 15 (DEM). *A data encapsulation mechanism (DEM)* DEM *is described by a (efficiently samplable) keyspace* $\mathcal{K}$ *and the two algorithms* $\mathsf{D}, \mathsf{D}^{-1}$*:*

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

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

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 $\mathsf{IND\text{-}RCCA}$ security for DEMs. It is described in Definition 16.

Definition 16. *The advantage of an adversary* $\mathcal{A}$ *against the* $\mathsf{IND\text{-}RCCA}$ *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. 9.*

```
Game IND-RCCA for DEM

Exp$^{IND-RCCA}_{DEM,b}(\mathcal{A})$
    $k \overset{\$}{\leftarrow} \mathcal{K}$
    $(St, m_0^*, m_1^*) \overset{\$}{\leftarrow} \mathcal{A}^{\mathrm{Dec}(\cdot),\mathrm{Enc}(\cdot)}$
    $c^* \overset{\$}{\leftarrow} \mathsf{D}(k, m_b)$
    return $\mathcal{A}^{\mathrm{Dec}(\cdot),\mathrm{Enc}(\cdot)}(St, c^*)$

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

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

Fig. 9: $\mathsf{IND\text{-}RCCA}$ security for DEMs.

A.5 Message Authentication Codes (MAC)

Message authentication codes are defined in Definition 17.

Definition 17. *A message authentication code* $\mathsf{MAC} = (\mathsf{MAC.tag}, \mathsf{MAC.vrf})$ *consist of a keyspace* $\mathcal{K}$ *and the following two algorithms:*

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

$\mathsf{MAC.vrf}(k, m, \mathsf{tag}) \overset{\$}{\rightarrow} \{0,1\}$*: The verification algorithm takes a key* k*, a message* m *and a tag* tag *and outputs either* 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*($\mathsf{EUF\text{-}CMA}$).

Definition 18. *A mac* MAC *is* $\mathsf{EUF\text{-}CMA}$ *secure, if for all PPT adversaries* $\mathcal{A}$ *the advantage*

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

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* $\mathsf{MAC.vrf}$ *algorithm on the two inputs with* k*.*

Additionally, we need a property called *Key Committing*, i.e. it should be hard to find two different keys for which a tag authenticates a message. This is again a symmetric analogue to *Exclusive Ownership* with the same reasoning for the different name, i.e. ownership of a symmetric key isn't a reasonable concept. We define it formally in Definition 19

Definition 19 (Key Commiting). *Let* $\mathsf{MAC} = (\mathsf{MAC.tag}, \mathsf{MAC.vrf})$ *be a MAC with keyspace* $\mathcal{K}$*. We define the advantage of an adversary* $\mathcal{A}$ *in the* Key Committing *game for* MAC *as*

$$\mathrm{Adv}^{\mathsf{KC}}_{\mathsf{MAC}}(\mathcal{A}) = Pr\left[\begin{array}{c} \mathsf{MAC.vrf}(k_1^*, m^*, \mathsf{tag}^*) = 1 \wedge \\ \mathsf{MAC.vrf}(k_2^*, m^*, \mathsf{tag}^*) = 1 \wedge \\ k_1^* \neq k_2^* \end{array}\middle|\ (k_1^*, k_2^*, m^*, \mathsf{tag}^*) \xleftarrow{\$} \mathcal{A}\right].$$

Remark 6. In general, $\mathsf{EUF\text{-}CMA}$ security does not imply KC security. This can easily be seen by considering a MAC which is $\mathsf{EUF\text{-}CMA}$ secure but ignores the last bit of its key. However, popular constructions like HMAC are naturally KC secure (as long as the hash function is collision resistant).

Additionally, any $\mathsf{EUF\text{-}CMA}$ secure MAC is KC secure in the random oracle model by tagging $(H(k)||m)$ instead of only m.

A.6 Huffman Trees

The Huffman Code[44] is an optimal prefix-free code, i.e. no codeword is a prefix of another code word and the weighted sum of all codewords is minimal. It is defined over a list of words X, their (relative) weights w and an alphabet C of size m. The Huffman code is represented by an m-ary tree, where each node is labelled by a symbol from the alphabet C and encoding a word is done by following the path from the root to the leaf corresponding to the word. Huffman codes are optimal, if the frequencies of the encoded symbols are powers of the tree arity. We will limit the definition to binary huffman trees. The construction algorithm is described in Fig. 10.

A.7 Key Committing Signatures

We define the notion of *Key Committing* for Reducible Signature. Intuitively, it should not be possible for an adversary to find different symmetric keys, such that a signature is valid under both keys.[16] The notion is similar in nature to notions of *exclusive ownership* [23], however the notion of ownership does not reasonably apply to symmetric keys.

Definition 20 (Reducible Key Committing). *Let* HRS *be a hedged Reducible Signature scheme. We define the advantage of an adversary* $\mathcal{A}$ *in the* Reducible Key Commiting(RKC) *experiment as*

$$\mathrm{Adv}^{\mathsf{RKC}}_{\mathsf{HRS}}(\mathcal{A}) = \Pr\left[\begin{array}{c} (\vec{m}, \sigma, k_1, \mathsf{RPC}) = Q[i] \wedge \\ \sigma' = \mathsf{Reduce}(\mathsf{vk}, \sigma, \vec{m}, \mathsf{rp}) \wedge \\ \mathsf{Vrfy}(\mathsf{vk}, k_2, \mathsf{rp}(\vec{m}), \mathsf{rp}, \sigma') = 1 \wedge \\ k_1 \neq k_2 \end{array}\middle|\begin{array}{c} (\mathsf{vk}, \mathsf{sk}) \xleftarrow{\$} \mathsf{KeyGen} \\ (k_2, \mathsf{rp}, i) \xleftarrow{\$} \mathcal{A}^{\mathbf{Sign}(\mathsf{sk},\cdot)}(\mathsf{vk}) \end{array}\right],$$

where the **Sign** *oracle, on input a tuple containing a message vector* $\vec{m}$*, a symmetric key* k *and a reduction class* RPC*, generates a signature* σ *using* sk*, adds* $(\vec{m}, \sigma, k, \mathsf{RPC})$ *to the list* Q *and outputs* σ*.*

[16] Looking ahead, this property prevents the following attack: A is in an epoch E where the symmetric HRS key is leaked (during corruption of B) and sends a message authenticated with her secure asymmetric key. Then, the adversary can deliver A's message to D in an epoch E', hence making C jump around the history graph. This attack is prevented because messages sent from E and delivered to E' are authenticated under different symmetric keys from the key schedules of E and E'.

Algorithm Huffman

```
Huffman($\vec{X}, \vec{w}$)
  Let $T$ be an empty binary tree
  for $x_i \in \vec{X}$ do
      $T$.nodes +← $x_i$
      $x_i$.weight $= w_i$

  $L = sort(T.\text{nodes})$      // ascending by weight and depth in $T$
  while $|L| \neq 1$ do

      $[x_1, x_2] = L[1:2]$      // first 2 elements in $L$
      $V$ +← $l_{new}$
      $l_{new}$.weight $= \sum_{i=1}^{q} w_i$
      for $i \in [2]$ do
          $x_i$.parent $:= l_{new}$
          $x_i$.label $= i$
      $L = L \setminus \{x_1, x_2\}$
      $L$ +← $l_{new}$

      $L = sort(L)$      // ascending by weight and depth in $T$
  return $T$
```

Fig. 10: Construction of a binary Huffman tree on word vector $\vec{X}$ with weights $\vec{w}$.

B Nominal Groups

We recall the definition of and parameters for nominal groups from [1].

Definition 21 (Nominal Group). *A nominal group $\mathcal{N} = (\mathcal{G}, g, p, \mathcal{E}_H, \mathcal{E}_U, \mathsf{exp})$ 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 $\mathsf{exp} : \mathcal{G} \times \mathbb{Z} \to \mathcal{G}$. We write X^y as shorthand for $\mathsf{exp}(X, y)$ 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 1 lists the parameters for these nominal groups.

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 1: Statistical parameters of NIST curves and nominal group curves.

For a more detailed explanation of these values, see [1]. 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 1.

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

Definition 22 (Double-Sided Strong Diffie-Hellman Assumption). *Let $\mathcal{N} = (\mathcal{G}, g, p, \mathcal{E}_H, \mathcal{E}_U, \mathsf{exp})$ be a nominal group. We define the advantage of an algorithm $\mathcal{A}$ in solving the* Double-Sided Strong Diffie-Hellman problem(DSSDH) *with respect to $\mathcal{N}$ as*

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

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 7. 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 5. *Let $\mathcal{N} = (\mathcal{G}, g, p, \mathcal{E}_H, \mathcal{E}_U, \mathsf{exp})$ be a nominal group. If the* DSSDH *assumption holds relative to $\mathcal{N}$ and* DEM *is an* $\mathsf{IND\text{-}RCCA}$ *secure DEM, then* mmPKE *from Fig. 2 is* $\mathsf{mmIND\text{-}RCCA}$ *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* $\mathsf{IND\text{-}RCCA}$ *of* DEM *respectively, s.t. for all adversaries $\mathcal{A}$ against the* $\mathsf{mmIND\text{-}RCCA}$

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

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 1 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.

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 1.

C Missing Proofs

C.1 Proof of Theorem 1

Theorem 1. *Let $\mathbb{G}$ be a group of prime order p with generator g, let* DEM *be a data encapsulation mechanism and let* $\mathsf{mmPKE} = \mathsf{DH\text{-}mmPKE}[\mathbb{G}, g, p, \mathsf{DEM}, \mathsf{Hash}]$. *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 such that*

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

where Hash *is modeled as a random oracle, e is the Euler number, n is the length of the encrypted challenge vector, and q_{d_1}, q_c and q_h are the number of queries to the oracles $\mathbf{Dec}_1$, $\mathbf{Cor}$ and the random oracle, respectively.*

Proof. We define n hybrids G_0 through G_n, where G_0 is identical to $\mathrm{Exp}^{\mathsf{mmIND\text{-}RCCA}}_{\mathsf{mmPKE},N,0}$, G_n is identical to $\mathrm{Exp}^{\mathsf{mmIND\text{-}RCCA}}_{\mathsf{mmPKE},N,1}$ 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.

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

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

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

Lemma 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 $\mathcal{A}$*

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

Combining the two lemmas and the remark yields Corollary 1 and the theorem follows from a standard hybrid argument over the games G_i.

Corollary 1. *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 $\mathcal{A}$*

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

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

So all that is left is proving the lemmas.

We will start by proving Lemma 1. Consider the formal definition of $G_{i,1}$ in Fig. 11.

Next, we describe adversary $\mathcal{B}_1$ against the DSSDH assumption in Fig. 12 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,

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

Game $G_{i,1}$

```
Game $G_{i,1}$
  for $i \in [N]$ do
      $(\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{KG}$
  $\mathsf{Corrupted} \leftarrow \varnothing, \mathsf{HL} \leftarrow \varnothing$
  $(\vec{\mathsf{ek}}^*, \vec{m}_0^*, \vec{m}_1^*, st) \leftarrow \mathcal{A}_1^{\mathrm{Dec}_1,\mathrm{Cor},\mathrm{H}}(g, \mathsf{ek}_1, \ldots, \mathsf{ek}_N)$
  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. $\forall j \in [l] : \vec{m}_0[i_j] \neq \vec{m}_1[i_j] \wedge \hat{\mathsf{ek}}_{i_j} \in \vec{\mathsf{ek}}$
  req $|\vec{m}_0^*| = |\vec{m}_1^*| = |\vec{\mathsf{ek}}^*| = n$
  $r \xleftarrow{\$} \mathbb{Z}_p, c_0^* \leftarrow g^r$
  for $1 \leq j \leq n$ do
      $K_j \leftarrow H(\mathsf{ek}_j^r, \mathsf{ek}_j, j)$
  $K^* \xleftarrow{\$} \mathcal{K}$
  $K_i \leftarrow K^*$
  for $1 \leq j \leq i$ do
      $c_j^* = \mathsf{D}(K_j, \vec{m}_1^*[j])$
  for $i \leq j \leq n$ do
      $c_j^* = \mathsf{D}(K_j, \vec{m}_0^*[j])$
  $c^* \leftarrow (c_0^*, \ldots, c_n^*)$
  $b \leftarrow \mathcal{A}^{\mathrm{Dec}_2,\mathrm{Cor},\mathrm{H}}(c^*, st)$
  req $\forall j \in [n] : \vec{\mathsf{ek}}^*[j] \notin \{\mathsf{ek} : i \in [N] \setminus \mathsf{Corrupted}\} \implies m_0^*[j] = m_1^*[j]$
  return $b$
```

```
Oracle Dec$_1(i, (c_0, c_i))$
  req $i \in [N]$
  $K \leftarrow H(c_0^{\mathsf{dk}_i}, \mathsf{ek}_i, i)$
  return $\mathsf{D}^{-1}(K, c_i)$

Oracle Dec$_2(j, c = (c_0, (c_k, k)))$
  req $j \in [N]$
  $K \leftarrow H(c_0^{\vec{\mathsf{dk}}[j]}, \vec{\mathsf{ek}}[j], k)$
  if $c_0 = c_0^* \wedge \vec{\mathsf{ek}}^*[k] = \vec{\mathsf{ek}}[j] \wedge i \leq l$ then
      $K \leftarrow K_j^*$
  $m \leftarrow \mathsf{D}^{-1}(K, c_k)$
  if $\exists k : \vec{\mathsf{ek}}^*[j] = \mathsf{ek}_k \wedge m \in \{\vec{m}_0^*[j], \vec{m}_1^*[j]\}$
  then
      return test
  else
      return $m$

Oracle Cor$(i)$
  req $i \in [N]$
  $\mathsf{Corrupted} \mathrel{+}\!\!\leftarrow i$
  return $\mathsf{dk}_i$

Oracle H$(Z, W, i)$
  if $\mathsf{HL}[Z, W, i] = \perp$ then
      $\mathsf{HL}[Z, W, i] \xleftarrow{\$} \mathcal{K}$
  return $\mathsf{HL}[Z, W, i]$
```

Fig. 11: Description of the hybrid $G_{i,1}$

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 $\mathsf{ek}_{i_1}, \ldots, \mathsf{ek}_{i_{q_C}}$ $b[i_j] = 0$ 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 Lemma 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.

C.2 Proof of Theorem 2

Theorem 2. *Let* Huf-wAcc[Hash] *denote the accumulator from Fig. 3 instantiated with a function* Hash : $\{0,1\}^* \to \{0,1\}^\kappa$*. For any adversary $\mathcal{A}$, there exists an adversary $\mathcal{B}$ such that*

$$\mathrm{Adv}^{\mathsf{UF}}_{\mathsf{Huf\text{-}wAcc[Hash]}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{CR}}_{\mathsf{Hash}}(\mathcal{B}).$$

Proof. Let $\mathcal{A}$ be any adversary against the unforgeability of wAcc. We construct an adversary $\mathcal{B}$ against the collision resistance of Hash with roughly the same runtime and the same success probability as $\mathcal{A}$.

$\mathcal{B}$ simulates the unforgeability game for $\mathcal{A}$ by simply forwarding the description of its hash function. Upon receiving a successful forgery $(\vec{X}^*, \vec{w}^*, \pi^*, x^*)$, $\mathcal{B}$ recomputes the accumulator value $acc^* = \mathsf{Eval}(\vec{X}^*, \vec{w}^*)$, which rebuilds the Huffman tree τ and computes the hashes along its paths. Since we assume that $\mathcal{A}$ outputs a valid forgery, $\mathsf{Vrfy}(acc^*, x^*, \pi^*) = 1$ but $x^* \notin \vec{X}^*$. Let $\hat{h}_1, \ldots \hat{h}_\ell$ denote the hashes computed during Vrfy. Since Vrfy returns 1, we have $\hat{h}_\ell = \tau.\mathsf{root}.\mathsf{label}$. Since also$x^* \notin \vec{X}^*$, there exists a node v in τ and i s.t. $\hat{h}_i = v.\mathsf{label}$ and either v is a leaf or for $v_l, v_r \in v.\mathsf{children}$: $\hat{h}_{i-1} \neq v_l.\mathsf{label} \wedge \hat{h}_{i-1} \neq v_r.\mathsf{label}$. If v is an internal node, then $\mathcal{B}$ outputs $(\text{'}int\text{'}, v_l.\mathsf{label}, v_r.\mathsf{label}), (\text{'}int\text{'}, \hat{h}_{i-1}, \pi[i])$ (resp. $(\text{'}int\text{'}, v_l.\mathsf{label}, v_r.\mathsf{label}), (\text{'}int\text{'}, \pi[i], \hat{h}_{i-1})$) as its forgery. If v is the leaf corresponding to $x_i \in \vec{X}^*$,

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

Adversary $\mathcal{B}_1(\mathbb{G}, p, g, U, V)$
  $\mathsf{Corrupted} \leftarrow \varnothing, \mathsf{HL} \leftarrow \varnothing, \mathsf{DL} \leftarrow \varnothing$
  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
  $\mathsf{Phase} \leftarrow 1$
  $(\vec{m}_0^*, \vec{m}_1^*, \vec{\mathsf{ek}}^*, st) \xleftarrow{\$} \mathcal{A}^{\mathsf{H},\mathsf{D}_1,\mathsf{Cor}}(\mathsf{ek}_1, \ldots, \mathsf{ek}_N)$
  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
          $\mathsf{HL}[U^{\alpha_j}, \mathsf{ek}^*[j], j] \leftarrow K_j$
  $\mathsf{Phase} \leftarrow 2$
  $b' \xleftarrow{\$} \mathcal{A}^{\mathsf{H},\mathsf{D}_2,\mathsf{Cor}}(c_0^*, \ldots, c_n^*, st)$
  return $\perp$

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 $\mathsf{Phase} = 1 \wedge \mathbf{O}_u(W, Z) = 1$ then
      $\mathsf{Bad}_2 \leftarrow 1$
      abort
  if $\mathsf{Phase} = 2 \wedge j \in [n] \wedge W = \mathsf{ek}^*[j] \wedge \mathbf{O}_u(W, Z) =$
  $1 \wedge j > i$ 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 $\mathsf{Phase} = 1 \wedge c_0 = U$ 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 $m$

Oracle $\mathsf{Cor}(j)$
  $\mathsf{Corrupted} \mathrel{+}\leftarrow j$
  if $b[j] = 1$ then
      $\mathsf{Bad}_3 \leftarrow True$
      abort
  return $\alpha_j$
```

Fig. 12: Description of adversary $\mathcal{B}_1$ from Theorem 1.

then $\mathcal{B}$ outputs $(\textit{'leaf'}, x_i)$, $(\textit{'int'}, \hat{h}_{i-1}, \pi[i])$ (respectively, $(\textit{'leaf'}, x_i), (\textit{'int'}, \pi[i], \hat{h}_{i-1})$) as its forgery, if $i \neq 0$ or $(\textit{'leaf'}, x_i), (\textit{'leaf'}, x^*)$ otherwise. In either case, $\mathcal{B}$ finds a collision if $\mathcal{A}$ outputs a valid forgery. ■

C.3 Proof for Theorem 3

Theorem 3. *Let* $\mathsf{HRS} = \mathsf{BGM\text{-}HRS}[\mathsf{Hash}, \mathsf{Acc}, \mathsf{Sig}, \mathsf{MAC}]$ *denote the scheme from Fig. 5 instantiated with a hash* Hash*, a signature scheme* Sig*, a cryptographic accumulator* Acc *and a* MAC*. For any adversary* $\mathcal{A}$*, there exist adversaries* $\mathcal{B}_1$*,* $\mathcal{B}_2$ *and* $\mathcal{B}_3$*, and* $\mathcal{B}_1'$*,* $\mathcal{B}_2'$ *and* $\mathcal{B}_3'$*, all with roughly the same runtime as* $\mathcal{A}$*, s.t.*

$$\mathrm{Adv}^{\mathsf{AEUF\text{-}RCMA}}_{\mathsf{HRS}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{Sig}}(\mathcal{B}_1) + \mathrm{Adv}^{\mathsf{CR}}_{\mathsf{Hash}}(\mathcal{B}_2) + \mathrm{Adv}^{\mathsf{UF}}_{\mathsf{Acc}}(\mathcal{B}_3) \textit{ and}$$

$$\mathrm{Adv}^{\mathsf{SEUF\text{-}RCMA}}_{\mathsf{HRS}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{EUF\text{-}CMA}}_{\mathsf{MAC}}(\mathcal{B}_1') + \mathrm{Adv}^{\mathsf{CR}}_{\mathsf{Hash}}(\mathcal{B}_2') + \mathrm{Adv}^{\mathsf{UF}}_{\mathsf{Acc}}(\mathcal{B}_3').$$

Proof. The proof for Theorem 3 is analogue to Theorem 3, therefore we will only describe the latter. Let $\mathcal{A}_1$ be an adversary against the $\mathsf{AEUF\text{-}RCMA}$ security of HRS. We construct the adversaries $\mathcal{B}_1, \mathcal{B}_2, \mathcal{B}_3$ against the respective primitives as follows:

$\underline{\mathcal{B}_1 \text{ against } \mathsf{EUF\text{-}CMA} \text{ of } \mathsf{Sig}}$: $\mathcal{B}_1$ receives a verification key spk as input and has access to a signing oracle $\mathcal{O}_{\text{sign}}$. $\mathcal{B}_1$ forwards spk to $\mathcal{A}_1$ and simulates its signing queries by executing the regular signing algorithm with the exception of getting the signature by querying its signing oracle on the final hash and the tag $(h^{(i)}||\mathsf{tag}^{(i)})$ (for the i-th signing query) with the verification key and uses the symmetric keys provided by the adversary. Eventually, $\mathcal{A}_1$ outputs a (potentially reduced) forgery $(\vec{m}^*, \sigma^* = (\sigma'^*, \mathsf{tag}^*, acc^*(, \pi^*, h_i^*)), \mathsf{spk}^*, k^*, \mathsf{rp}^* = (\ell^*, *, *)$. $\mathcal{B}_1$ computes the hash $h^* = \mathsf{Hash}(\ell^*||acc||h_1^*)$ and outputs $(h^*||\mathsf{tag}^*), \sigma^*$ as its forgery.
We analyse $\mathcal{B}_1$'s winning probability: Since $\mathcal{A}_1$ outputs a valid forgery, $\vec{m}^*$ is distinct from all previous message vectors queried to the sign oracle. $\mathcal{A}_1$ controls the symmetric key k^*, so we can't assume that tag^* is fresh, so h^* has to be a new hash value in order for $\mathcal{B}_1$ to be successful. However for h^* to be a previous hash while $\vec{m}^*$ is different from all previous queries, $\mathcal{A}_1$ has to produce either a hash collision or an accumulator forgery. So $\mathcal{B}_1$ wins exactly if $\mathcal{B}_2$ and $\mathcal{B}_3$ don't win their respective games.

$\underline{\mathcal{B}_2 \text{ against collision resistance of } \mathsf{Hash}}$: $\mathcal{B}_2$ generates a signature keypair $\underline{(\mathsf{ssk}, \mathsf{spk}) \xleftarrow{\$} \mathsf{Sig.kg}}$ and then runs $\mathcal{A}_1$ on input spk. $\mathcal{B}_2$ answers signing queries using ssk and for all signing queries $(k^{(i)}, \vec{m}^{(i)}, \mathsf{RPC}^{(i)})$ records $(\vec{m}^{(i)}, \sigma^{(i)} = (\sigma'^{(i)}, \mathsf{tag}^{(i)}, acc^{(i)}), h^{(i)}, \ell^{(i)})$ in a list L, where $\sigma^{(i)}$ is the reducible signature and $h^{(i)}$ is the signed hash. When $\mathcal{A}_1$ outputs a forgery $(\vec{m}^*, \sigma^* = (\sigma'^*, \mathsf{tag}^*, acc^*(, \pi^*, h_{j^*}^*)), k^*, \mathsf{spk}^*, \mathsf{rp}^* = (\ell^*, i^*, j^*)$, $\mathcal{B}_2$ recomputes $h^* = \mathsf{Hash}(\ell^*, acc, \hat{h}^*)$ as in the verification algorithm and checks its list L for a corresponding pair $(\vec{m}, (\sigma', \mathsf{tag}, acc), h^*, \ell)$ with the same signed hash h^*. If none is found, $\mathcal{A}_1$ produced a signature forgery and $\mathcal{B}_2$ aborts (in this case, $\mathcal{B}_1$ would win). So assume there is such a pair.
If $acc = acc^*$ and $\ell = \ell^*$ and $\vec{m}[i] = \vec{m}^*[i]$ for $2 \leq i \leq \ell$, then $\mathcal{A}_1$ produced and accumulator forgery and $\mathcal{B}_2$ aborts. Note that $\mathcal{B}_3$ is successful in this case.
So assume that $acc \neq acc^*$ or $\hat{h}^* \neq \hat{h}$, i.e. the final hash in the chain differs. Then $(l, acc, \hat{h}), (\ell^*, acc^*, \hat{h}^*)$ is a collision in Hash and $\mathcal{B}_2$ wins.
Now assume that $acc = acc^*$ and $\hat{h}^* \neq \hat{h}$ but there exists an $i > 1$ s.t. $\vec{m}^*[i] \neq \vec{m}[i]$. Then there exists an $i^* \leq i$ s.t. $\mathsf{Hash}(\vec{m}[i^*], \hat{h}) = \mathsf{Hash}(\vec{m}^*[i^*], \hat{h}^*)$ but $\hat{h}^* \neq \hat{h}$ or $\vec{m}^*[i^*] \neq \vec{m}[i^*]$, which again yields a collision in Hash. Note that said i^* necessarily exists since we assume that $\hat{h}$ and $\hat{h}^*$ coincide before the computation of the signed hash h^*.

$\underline{\mathcal{B}_3 \text{ against unforgeability of } \mathsf{Acc}}$: $\mathcal{B}_3$ generates a signature key pair $(\mathsf{ssk}, \mathsf{spk}) \xleftarrow{\$} \mathsf{Sig.kg}$ and then calls $\mathcal{A}_1$ on input spk. Signing queries by $\mathcal{A}_1$ are answered honestly with the signature key ssk and for every signing query, $\mathcal{B}_3$ records the pairs $(\vec{m}^{(i)}, acc^{(i)})$ in a list L.
Eventually, $\mathcal{A}_1$ outputs a forgery $\vec{m}^* = (m_1^*, m_2^*, \ldots, m_{j^*+1}^*)$,
$\sigma^* = (\sigma'^*, \mathsf{tag}^*, acc^*, \pi^*, h_{j^*}^*), k^*, \mathsf{spk}^*, \mathsf{rp}^* = (\ell^*, i^*, j^*))$. We assume that $\mathcal{A}_1$ didn't output a signature forgery or found a hash collision, because in those cases, $\mathcal{B}_1$ and $\mathcal{B}_2$ win their respective games. Therefore, there exists an t^* s.t. $acc^{(t^*)} = acc^*$. Since no hash collision occurred, $\vec{m}^{(t^*)}[j] = \vec{m}^*[j]$ for $j \geq \ell^*$ but since $\mathcal{A}_1$ produced a valid forgery, $\vec{m}^{(t^*)}[i^*] \neq \vec{m}^*[i^*]$, because otherwise $\vec{m}^*$ would be a valid reduction from $\vec{m}^{(t^*)}$. $\mathcal{A}_1$'s forgery can only be valid if the accumulator proof verifies and then $\vec{m}^{(t^*)}, \vec{m}^*[i^*], \pi^*$ is a valid accumulator forgery.

It is easy to check that in every case one of the adversaries $\mathcal{B}_1$,$\mathcal{B}_2$, $\mathcal{B}_3$ aborts, another would win. So we can bound the success probability of $\mathcal{A}$ by the sum of their success probabilities.

For $\mathcal{A}_2$, we replace all generations of asymmetric signing keys with sampling a symmetric key and replacing the first adversary with one against the MAC, which works in exactly the same way.

Game mmOW-RCCA

```
$\underline{\mathrm{Exp}^{\mathsf{mmIND\text{-}RCCA}}_{\mathsf{mmPKE},N}(\mathcal{A} = (\mathcal{A}_1, \mathcal{A}_2))}$
  for $i \in [N]$ do
      $(\mathsf{ek}_i, \mathsf{dk}_i) \leftarrow \mathsf{mmPKE.KG}()$
  $\mathsf{Corrupted} \leftarrow \varnothing$
  $(\vec{\mathsf{ek}}, \vec{m}, S, st) \leftarrow \mathcal{A}_1^{\mathrm{Dec}_1,\mathrm{Cor}}(\mathsf{ek}_1, \ldots, \mathsf{ek}_N)$
  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 $\vec{m}[j] \leftarrow m^*$
  $m' \leftarrow \mathcal{A}^{\mathrm{Dec}_2,\mathrm{Cor}}(\mathsf{mmPKE.Enc}(\vec{\mathsf{ek}}, \vec{m}), st)$
  req $EK^* \subseteq \{\mathsf{ek}_i : i \in [N] \setminus \mathsf{Corrupted}\}$
  return $m^* = m'$
```

```
$\underline{\textbf{Oracle Dec}_1(i, c)}$
  req $i \in [N]$
  return $\mathsf{Dec}(\vec{\mathsf{dk}}_i, c)$

$\underline{\textbf{Oracle Cor}(i)}$
  req $i \in [N]$
  $\mathsf{Corrupted} \mathrel{+}\!\leftarrow i$
  return $\mathsf{dk}_i$
```

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

Fig. 13: Experiment defining mmOW-RCCA security of mmPKE schemes.

C.4 HRS is RKC secure

Theorem 6. *Let* MAC *be a* KC *secure mac. Then the Reducible Signature scheme described in Fig. 5 is* RKC *secure. Specifically for any adversary $\mathcal{A}$, there exists an adversary $\mathcal{B}$ with roughly the same runtime as $\mathcal{A}$, s.t.*

$$\mathrm{Adv}^{\mathsf{RKC}}_{\mathsf{HRS}}(\mathcal{A}) \leq \mathrm{Adv}^{\mathsf{KC}}_{\mathsf{MAC}}(\mathcal{B})$$

Proof. First, note that since the adversary outputs one message and signature, which has to be valid for two keys, the security of the accumulator isn't relevant for exclusive ownership security.

Let $\mathcal{A}$ be an adversary against the MS-sCEO security of HRS. We construct adversary $\mathcal{B}$ as follows: $\mathcal{B}$ samples a keypair $(\mathsf{vk}, \mathsf{sk})$ and runs $\mathcal{A}$ on input vk. Signing queries are answered using sk and the provided secret key. Eventually, $\mathcal{A}$ outputs its solution $(k_1^*, k_2, m^*, \sigma^*)$ with $\sigma^* = (\sigma', \mathsf{tag}^*, acc, \pi, h)$. $\mathcal{B}$ then computes h^* (the root hash that is tagged by tag^*) and outputs $(k_1^*, k_2^*, m^* := h^*, \mathsf{tag}^*)$ as its solution. It is easy to see that $\mathcal{B}$ is successful, if $\mathcal{A}$ is successful.

D One-Wayness Security of mmPKE

In this section, we define One-Wayness under Relaxed Chosen Ciphertext Attacks security of mmPKE schemes, 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. 13, 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's, 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 23 (mmOW-RCCA). *For a scheme* mmPKE *with message space $\mathcal{M}$, we define the advantage of an adversary $\mathcal{A}$ against* One-Wayness Against Replayable Chosen Ciphertext Attacks (mmOW-RCCA) *security of* mmPKE *as*

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

where $\mathrm{Exp}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE},N}(\mathcal{A})$ *is defined in Fig. 13.*

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

Theorem 7. *Let* mmPKE *be an mmPKE scheme with message space $\mathcal{M}$. For any adversary $\mathcal{A}$, there exists an adversary $\mathcal{B}$ such that*

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

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

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

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 $\frac{1}{\mathcal{M}}$. ■

E Details of the SAIK Protocol

SAIK is heavily based on ITK. We recall the whole protocol here for completeness. The major difference between SAIK and ITK is the use of mmPKE and HRS. There are also some smaller differences, such as the use of q-ary trees. Another small difference is that ITK ensures agreement on the "transcript hash", binding all past messages, while SAIK ensures agreement only on all past group modifications. Indeed, a transcript hash does not make sense for saCGKA, since parties see different messages.

E.1 Authenticated Key Service (AKS)

SAIK relies on an Authenticated Key Service (AKS) which authentically distributes so-called key packages (also called key bundles or pre-keys) used to add new members to the group without interacting with them. A key package should only be used once. For simplicity, we use an idealized AKS which guarantees that a fresh, authentic and honestly generated key package of any party is always available to any other party.

Formally, the AKS is modelled as the functionality $\mathcal{F}_{\text{AKS}}$ defined in Fig. 14. SAIK works in the `*pk`-hybrids model. This means that $\mathcal{F}_{\text{AKS}}$ 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 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.

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

Parameters: a key-package generation algorithm `*AKS-kgen()`.

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

Inputs

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

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

Fig. 14: The Authenticated Key service Functionality.

To conclude, we mention the most important aspects in which $\mathcal{F}_{\text{AKS}}$ differs from a more realistic AKS. First, in a typical implementation of an AKS, parties generate key packages themselves and upload them to an untrusted server, authenticated with long-term so-called identity keys. This means that a realistic attacker model is one where parties can be corrupted before they join, in which case the secrets for their key packages and long-term keys leak. This allows an active adversary to inject arbitrary key packages on their behalf. Such abilities are not considered in our model. However, we stress that we do consider attacks where the adversary injects on behalf of current group members messages that add parties with arbitrary key packages.

Further, $\mathcal{F}_{\text{AKS}}$ identifies key packages by public keys. Looking ahead, this means that a party adding id' has to send the whole public part of the package so that id' can identify it when it joins. In reality, this would be implemented by hashes.

E.2 High-level Description

Ratchet trees. The operation of SAIK, as well as of ITK, crucially relies on a data structure called ratchet trees. A ratchet tree τ is a tree where each leaf is assigned to a group member. A leaf stores information about its member's identity and their HRS key pair. Moreover, most non-root nodes in τ, and all leaves, have assigned an mmPKE key pair. Nodes without a key pair are called *blank*.

The protocol maintains the *tree invariant* saying that each group member knows all public keys in the tree, as well as the secret keys of the nodes on the path from their leaf to the root, and only those. This allows to efficiently encrypt messages to subgroups: If a node v is not blank, then a message m can be encrypted to all parties 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.

SAIK uses generalized q-ary ratchet trees, generalizing binary trees used by ITK (and all its variants). While in most applications $q = 2$ is optimal, choosing $q > 2$ is more efficient if SAIK is instantiated with (an mmPKE built from) a post-quantum secure mKEM [46].

Ratchet tree evolution. Each epoch change triggers a number of modifications in the ratchet tree τ. First, if the epoch removes a party id_t, then all keys known to id_t are removed. That is, all nodes on the path from id_t's leaf to the root are blanked. If instead the epoch adds a party id_t, then a new leaf is inserted into τ. The new leaf's HRS and mmPKE public keys are fetched from the AKS. Moreover, all nodes from the new leaf to the root are blanked, because for FS id_t should not know any secrets from before it joined.

Further, in order to achieve PCS, the party id_s creating the epoch refreshes all key pairs in τ for which it knows secret keys. Specifically, id_s generates a new HRS key pair and a sequence of mmPKE key pairs for all nodes $v_1, \ldots, v_t$ on the path from their leaf to the root.

To maintain the tree invariant, id_s must communicate the new secret key of v_i to all members in v_i's subtree, for each $i > 1$ (the leaf's secret is only known to id_s). To minimize the communication complexity, id_s generates the key pairs as follows:

1. Generate a sequence of *path secrets* $s_2, \ldots, s_t$: s_2 is random and $s_{i+1} = \mathsf{Hash}(s_i, \text{'path'})$.
2. Generate a fresh key pair for v_1. For each $i > 1$, generate the key pair of v_i using randomness $\mathsf{Hash}(s_i, \text{'rand'})$.

Now id_s encrypts each s_i to all nodes in τ which are in the resolution of v_i but not in the resolution of v_{i-1}. This way, a group member in the subtree of v_i learns $s_i, \ldots, s_t$ but not $s_2, \ldots, s_{i-1}$ and the tree invariant is maintained. The encryption is done efficiently using mmPKE. In particular, id_s sends out a single mmPKE ciphertext.

Key schedule. Apart from the ratchet tree, all group members store a number of shared secrets:

- The *application secret* — the group key exported to the E2E application
- The *membership key* — a symmetric HRS key used to authenticate all sent messages
- The *init key* — mixed in the next epoch's application secret for FS

When a new epoch is created, its secrets are derived as follows: First a fresh *commit secret* is derived from the last path secret sent by the epoch's creator. Then, the init and commit 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 of an epoch includes all relevant information about it, such as (the hash of) the ratchet tree (which includes the member set). Intuitively, the purpose of mixing it into the key schedule is ensuring that if parties are in different epochs with different contexts, then they derive independent epoch secrets.

Authenticating packets. When an id_s creates an epoch, it authenticates the data using HRS with the asymmetric key from its leaf and the current epoch's membership key. The reduction pattern class allows to authenticate a single individual mmPKE ciphertext and a subsequence of public keys.

Importantly, each receiver also verifies *which* individual mmPKE ciphertext they received. Without this check, the adversary could, without corrupting id_s, make one receiver id_r accept a ciphertext meant for a different receiver id'_r. Depending on mmPKE, id_r may decrypt a different but still valid looking ciphertext. In this case, the adversary carries out a successful attack against authenticity.

Observe that the adversary should not be able to impersonate id_s even if it corrupted id_r. If this is the case, the adversary knows the value id_r would decrypt. This breaks attempts to prevent the above attack by adding confirmation tags proving knowledge of the encrypted secrets.

Joining. If an id_s creates an epoch adding a new member id_t, then in order to join id_t needs two secrets: its path secret needed to maintain the tree invariant and the new epoch's joiner secret needed to compute the key schedule. 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. Both path and joiner secret are included in the single mmPKE ciphertext uploaded by id_s to the mailboxing service. They are encrypted under an mmPKE public key fetched from the AKS and used only for this purpose (after joining, id_t deletes the secret key).

E.3 Ratchet Trees

In this section, we formally define ratchet trees. A ratchet tree is a left-balanced q-ary tree, defined as follows.

Balanced trees.

Definition 24 (Left-Balanced Tree). *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 25 (Full Left-Balanced Tree). *For $q, n \in \mathbb{N}$, $\mathsf{LBT}_{q,n}$ is full if n is a power of q.*

Node labels. A ratchet tree has all tree-related labels defined in Sec. 3, as well as additional labels listed in Table 2. We also define a number of helper methods in Table 3. Observe that the *direct path* of a leaf consists of the (ordered list) of all nodes on the path from the leaf to the root, without the leaf itself. The *resolution* of a node v is the minimal set of descendant non-blank nodes that covers the whole sub-tree rooted at v, i.e., such that for every descendant u of v there exists node w in the resolution such that w is non-blank and w an ancestor of u

Adding leaves. Node indices $v.\mathsf{nodeIdx}$ in ratchet trees 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. 15 for an example. Operation of SAIK requires adding leaves to a ratchet tree in a way that preserves node indices. We describe the algorithm $\mathsf{addLeaf}(\tau, v)$ which inserts v into τ in this way.

$v.\mathsf{ek}$	The encryption (public) key of an mmPKE scheme.
$v.\mathsf{dk}$	The corresponding decryption (secret) key.
$v.\mathsf{vk}$	If $v.\mathsf{isleaf}$: the verification (public) key of an HRS scheme.
$v.\mathsf{sk}$	If $v.\mathsf{isleaf}$: the corresponding signing (secret) key of an HRS scheme.
$v.\mathsf{unmergedLvs}$	The set of indices of those leaves below v for which the party id does not know $v.\mathsf{sk}$.
$v.\mathsf{id}$	If $v.\mathsf{isleaf}$: the identity associated with that leaf.

Table 2: Protocol-related labels of a ratchet-tree node v.

$\tau.\mathsf{clone}()$	Returns and (independent) copy of τ.
$\tau.\mathsf{public}()$	Returns a copy of τ for which all private labels ($v.\mathsf{sk}$) are set to $\perp$.
$\tau.\mathsf{roster}()$	Returns the identities of all parties in the tree.
$\tau.\mathsf{leaves}()$	Returns the list of all leaves in the tree, sorted from left to right.
$\tau.\mathsf{leafof}(\mathsf{id})$	Returns the leaf v for which $v.\mathsf{id} = \mathsf{id}$ (or $\perp$ if no such v exists).
$\tau.\mathsf{getLeaf}()$	Returns the first leaf v from the left for which $\neg v.\mathsf{inuse}()$. If no such leaf exists, adds a new leaf using $\mathsf{addLeaf}$ and returns that.
$\tau.\mathsf{directPath}(v)$	Returns the direct path, excluding the leaf, as an ordered list from the leaf to root.
$\tau.\mathsf{isInSubtree}(u, v)$	Returns true if u is in v's subtree.
$\tau.\mathsf{lca}(u, v)$	Returns the lowest common ancestor of the two leafs.
$\tau.\mathsf{blankPath}(v)$	Calls $u.\mathsf{blank}()$ on all $u \in \tau.\mathsf{directPath}(\mathsf{leaf})$.
$\tau.\mathsf{mergeLeaves}(v)$	Sets $u.\mathsf{unmergedLvs} \leftarrow \varnothing$ for all $u \in \tau.\mathsf{directPath}(v)$
$\tau.\mathsf{unmergeLeaf}(v)$	Sets $u.\mathsf{unmergedLvs} \mathrel{+}\!\leftarrow v$ for all u returned by $\tau.\mathsf{directPath}(v)$
$v.\mathsf{inuse}()$	Returns `false` iff all labels are $\perp$.
$v.\mathsf{blank}()$	Sets all labels to $\perp$.
$v.\mathsf{resolution}()$	Return $\begin{cases} (v) \mathbin{+\!\!+} v.\mathsf{unmergedLvs} & \text{if } v.\mathsf{inuse}() \\ \mathsf{concatChildResolution}(v) & \text{else if } \neg v.\mathsf{isleaf} \\ () & \text{else,} \end{cases}$ where $\mathsf{concatChildResolution}(v) = v.\mathsf{children}[1].\mathsf{resolution}() \mathbin{+\!\!+} \cdots \mathbin{+\!\!+}. v.\mathsf{children}[n].\mathsf{resolution}()$
$v.\mathsf{resolvent}(u)$	For a descendant u of v, returns the (unique) node in $v.\mathsf{resolution}() \setminus (v)$ which is an ancestor of u.

Table 3: Helper methods defined on the ratchet tree τ and its nodes v.

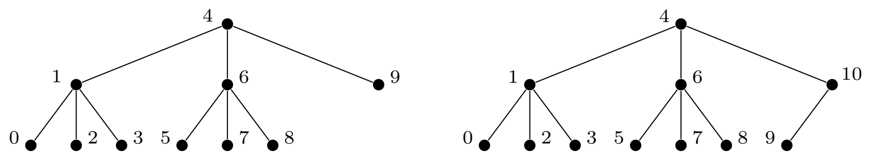


Fig. 15: The trees $\mathsf{LBT}_{3,7}$ (left) and $\mathsf{LBT}_{3,8}$ (right) with node indices.

Definition 26 (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 $r.\mathsf{children}$ *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 $r.\mathsf{children}$ *s.t. its subtree τ_u is not full. Let $\tau' = \tau$ except τ_u is replaced by* $\mathsf{addLeaf}(\tau_u, v)$.

Lemma 3. $\tau = \mathsf{LBT}_{q,n} \implies \mathsf{addLeaf}(\tau, v) = \mathsf{LBT}_{q,n+1}$.

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 $n \geq q$ and assume the statement holds for all $k < n$. Let r be the root of τ and let $\mathsf{max\text{-}pow}(n) = \max\{q^p + 1 : p \in \mathbb{N} \wedge q^p < n\}$.

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

Else, $\mathsf{max\text{-}pow}(n) = \mathsf{max\text{-}pow}(n + 1)$ (this holds since $n \geq q$). Moreover, it is easy to see that only the last node in $r.\mathsf{children}$ 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 $\mathsf{LBT}_{q,\mathsf{max\text{-}pow}(n)} = \mathsf{LBT}_{q,\mathsf{max\text{-}pow}(n+1)}$.
– 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 $x < \mathsf{max\text{-}pow}(n)$, 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. ■

E.4 Protocol State

The state of SAIK consists of a number of variables, outlined in Table 4. 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 $\gamma.\mathsf{leaf}$. This means that they agree on the secrets $\gamma.\mathsf{appSec}$ and $\gamma.\mathsf{initSec}$, as well as on the public context, computed by the helper method in Table 4.

$\gamma.\mathsf{grpId}$	The identifier of the group.
$\gamma.\tau$	The ratchet tree.
$\gamma.\mathsf{leaf}$	The party's leaf in τ.
$\gamma.\mathsf{treeHash}$	A hash of the public part of τ.
$\gamma.\mathsf{lastAct}$	The last modification of the group state and the user who initiated it.
$\gamma.\mathsf{appSec}$	The current epoch's CGKA key.
$\gamma.\mathsf{initSec}$	The next epoch's init secret.
$\gamma.\mathsf{membKey}$	The next epoch's membership secret for authenticating messages.

$\gamma.\mathsf{groupCtxt}()$	Returns $(\gamma.\mathsf{grpId}, \gamma.\mathsf{treeHash}, \gamma.\mathsf{lastAct})$.

Table 4: The protocol state and the helper method for computing the context.

E.5 Protocol Algorithms

The protocol algorithms are defined in Fig. 17 with some additional helper functions outsourced to Fig. 18.

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()
    $\gamma.\mathsf{leaf} \leftarrow \gamma.\tau.\mathsf{leaves}[0]$
    $(\gamma.\mathsf{leaf.vk}, \gamma.\mathsf{leaf.sk}) \leftarrow \mathsf{RS.KeyGen}()$

Inputs
Input (Send, act), $\mathsf{act} \in \{\mathsf{up}, \mathsf{rem\text{-}id}_t, \mathsf{add\text{-}id}_t\}$
  req $\gamma \neq \bot$
  // In case of add, fetch $\mathsf{id}_t$'s keys from AKS and store them in act
  (AKS runs *AKS-kgen).
  if $\mathsf{act} = \mathsf{add\text{-}id}_t$ then
    $(\mathsf{ek}_t, \mathsf{vk}_t, \mathsf{ek}'_t) \leftarrow$ query (GetPk, $\mathsf{id}_t$) to $\mathcal{F}_{\mathrm{KS}}$
    $\mathsf{act} \leftarrow \mathsf{add\text{-}id}_t\text{-}(\mathsf{ek}_t, \mathsf{vk}_t, \mathsf{ek}'_t)$
  // 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. In
  case of add, also encrypt the joiner secret for $\mathsf{id}_t$.
  if $\mathsf{act} \in \{\mathsf{up}, \mathsf{rem\text{-}id}_t\}$ then
    $\mathit{Ctxt} \leftarrow$ *encrypt$(\gamma', \mathsf{pathSecs}, \bot, \bot, \bot)$
  else if $\mathsf{act} = \mathsf{add\text{-}id}_t\text{-}(\mathsf{ek}_t, \mathsf{vk}_t, \mathsf{ek}'_t)$ then
    $\mathit{Ctxt} \leftarrow$ *encrypt$(\gamma', \mathsf{pathSecs}, \mathsf{id}_t, \mathsf{ek}'_t, \mathsf{joinerSec})$
  // Sign data under current epoch's secrets.
  $\mathsf{updEKs} \leftarrow ((\gamma'.\mathsf{leaf.ek}, \gamma'.\mathsf{leaf.vk}))$
  $\mathsf{updEKs} \leftarrow \mathsf{updEKs} \mathbin{+\!\!+} (v.\mathsf{ek} : v \in \gamma'.\tau.\mathsf{directPath}(\gamma'.\mathsf{leaf}))$
  $(\vec{\mathsf{tbs}}, \mathsf{RPC}) \leftarrow$ *to-be-signed$(\mathit{Ctxt}, \mathsf{updEKs}, \mathsf{act})$
  $\mathsf{sig} \leftarrow \mathsf{RS.Sign}(\gamma.\tau.\mathsf{leafof}(\mathsf{id}).\mathsf{sk}, \gamma.\mathsf{membKey}, \vec{\mathsf{tbs}}, \mathsf{RPC})$
  $\gamma \leftarrow \gamma'$
  // In case of add, send additional data for $\mathsf{id}_t$.
  if $\mathsf{act} = \mathsf{add\text{-}id}_t\text{-}(\mathsf{ek}_t, \mathsf{vk}_t, \mathsf{ek}'_t)$ then
    $\mathsf{welcomeData} \leftarrow (\gamma.\mathsf{grpId}, \gamma.\tau.\mathsf{public}(), \mathsf{ek}'_t)$
    return $(\mathsf{id}, \mathsf{act}, \mathit{Ctxt}, \mathsf{updEKs}, \mathsf{sig}, \mathsf{welcomeData})$
  return $(\mathsf{id}, \mathsf{act}, \mathit{Ctxt}, \mathsf{updEKs}, \mathsf{sig})$

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

Input (Receive, $(\mathsf{id}_s, \mathsf{rem}, \mathsf{sig}')$)
// Receiver is removed.
  $\vec{\mathsf{tbv}} \leftarrow ((\mathsf{id}_s, \mathsf{rem\text{-}id}, \gamma.\mathsf{grpId}))$
  // Check if removing allowed and compute the reduction pattern
  $\mathsf{rp} = (\ell, 0, 1)$.
  try $\gamma' \leftarrow$ *apply-act$(\gamma.\mathsf{clone}(), \mathsf{id}_s, \mathsf{rem\text{-}id})$
  $\ell \leftarrow$ *weights$(\gamma', \mathsf{id}_s)$
  $\mathsf{vk} \leftarrow \gamma.\tau.\mathsf{leafof}(\mathsf{id}_s).\mathsf{vk}$
  req $\mathsf{RS.Vrfy}(\mathsf{vk}, \gamma.\mathsf{membKey}, \vec{\mathsf{tbv}}, (\ell, 0, 1), \mathsf{sig}')$
  $\gamma \leftarrow \bot$
  return $(\mathsf{id}_s, \mathsf{rem\text{-}id})$

Input (Receive, $(\mathsf{id}_s, \mathsf{act}, \mathit{ctxt}, \mathsf{updEKs}', \mathsf{sig}')$)
// Receiver is a member.
  try $\gamma' \leftarrow$ *apply-act$(\gamma.\mathsf{clone}(), \mathsf{id}_s, \mathsf{act})$
  // Get the expected reduction pattern using the new state.
  try $\mathsf{rp} \leftarrow$ *my-reduction-pattern$(\gamma'.\tau, \mathsf{id}_s)$
  $\vec{\mathsf{tbv}} \leftarrow (\mathit{ctxt}) \mathbin{+\!\!+} ((\mathsf{id}_s, \mathsf{act}, \gamma.\mathsf{grpId})) \mathbin{+\!\!+} \mathsf{updEKs}'$
  $\mathsf{vk} \leftarrow \gamma.\tau.\mathsf{leafof}(\mathsf{id}_s).\mathsf{vk}$
  req $\mathsf{RS.Vrfy}(\mathsf{vk}, \gamma.\mathsf{membKey}, \vec{\mathsf{tbv}}, \mathsf{rp}, \mathsf{sig}')$
  // Transition to next epoch.
  try $\gamma \leftarrow$ *transition$(\gamma', \mathit{ctxt}, \mathsf{updEKs}', \mathsf{id}_s, \mathsf{act})$
  // Compute the output.
  if $\mathsf{act} = \mathsf{add\text{-}id}_t\text{-}(\mathsf{ek}_t, \mathsf{vk}_t)$ then return $(\mathsf{id}_s, \mathsf{add\text{-}id}_t)$
  else return $(\mathsf{id}_s, \mathsf{act})$

Input (Receive, $(\mathsf{id}_s, \mathsf{act}, \mathit{ctxt}_1, \mathit{ctxt}_2, \mathsf{welcomeData})$))
// Receiver joins.
  req $\gamma = \bot$
  parse $\mathsf{grpId}, \tau, \mathsf{ek}' \leftarrow \mathsf{welcomeData}$
  $\gamma \leftarrow$ *new-state
  $(\gamma.\mathsf{grpId}, \gamma.\tau, \gamma.\mathsf{lastAct}) \leftarrow (\mathsf{grpId}, \tau, (\mathsf{id}_s, \mathsf{add\text{-}id}))$
  $v \leftarrow \gamma.\tau.\mathsf{leafof}(\mathsf{id})$
  try $(v.\mathsf{dk}, v.\mathsf{sk}, \mathsf{dk}') \leftarrow$ query GetSk$((v.\mathsf{ek}, v.\mathsf{vk}, \mathsf{ek}'))$ to $\mathcal{F}_{\mathrm{KS}}$
  $\gamma \leftarrow$ *set-tree-hash$(\gamma)$
  try $\gamma \leftarrow$ *populate-secrets$(\gamma, \mathsf{dk}', \mathit{ctxt}_j, \mathit{ctxt}_{n+1}, \mathsf{id}_s)$
  return $(\gamma.\tau.\mathsf{roster}(), \mathsf{id}_s)$
```

Fig. 16: The algorithms of SAIK.

SAIK: Helpers for encryption and authentication

```
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 len$(L)$ do
    $(i, v) \leftarrow L[j]$
    $\vec{m} \mathbin{+\!\!\leftarrow} \mathsf{pathSecs}[i]$
    if $\mathsf{id}_t \neq \bot \wedge v = \gamma'.\tau.\mathsf{leafof}(\mathsf{id}_t)$ then $\vec{\mathsf{ek}} \mathbin{+\!\!\leftarrow} \mathsf{ek}'_t$
    else $\vec{\mathsf{ek}} \mathbin{+\!\!\leftarrow} \vec{v}.\mathsf{ek}$
  if $\mathsf{id}_t \neq \bot$ then
    $\vec{m} \mathbin{+\!\!\leftarrow} \mathsf{joinerSec}$
    $\vec{\mathsf{ek}} \mathbin{+\!\!\leftarrow} \mathsf{ek}'_t$
  return $(\mathsf{mmPKE.Enc}(\vec{\mathsf{ek}}, \vec{m}))$

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 $\mathsf{mmPKE.Dec}(v.\mathsf{dk}, \mathit{ctxt})$

helper *to-be-signed$(\gamma', \mathit{Ctxt}, \mathsf{updEKs}, \mathsf{act})$
  $\vec{w} \leftarrow$ *weights$(\gamma'.\tau, \mathsf{id})$
  $\vec{\mathsf{tbs}} \leftarrow ()$
  for $j = 1$ to $|\vec{w}|$ do
    $\vec{\mathsf{tbs}} \mathbin{+\!\!\leftarrow} \mathsf{mmPKE.Ext}(\mathit{Ctxt}, j)$
  $\vec{\mathsf{tbs}} \mathbin{+\!\!\leftarrow} (\mathsf{id}, \mathsf{act}, \gamma.\mathsf{grpId})$
  $\vec{\mathsf{tbs}} \mathbin{+\!\!\leftarrow} \mathsf{updEKs}$
  return $(\vec{\mathsf{tbs}}, \mathsf{RPC}^{\mathrm{SAIK}}_{|\vec{w}|,\vec{w}})$

helper *my-reduction-pattern$(\tau', \mathsf{id}_s)$
  $L \leftarrow$ *rcvrs-of-path-secs$(\tau', \mathsf{id}_s)$
  for $j = 1$ to len$(L)$ do
    $(i, v) \leftarrow L[j]$
    if $\gamma'.\tau.\mathsf{isInSubtree}(\tau'.\mathsf{leafof}(\mathsf{id}), v)$ then
      // We want the $j$-th ciphertext out of len$(L)$ and the first $i+1$ items on
      the prefix list: the aux data, the leaf ek and $i-1$ ek's on $\mathsf{id}_s$'s direct path.
      return (len$(L), j, i)$
  return $\bot$

helper *rcvrs-of-path-secs$(\tau', \mathsf{id}_s)$
  // Returns a list of tuples $(i, v)$, denoting that when $\mathsf{id}_s$ commits in $\tau'$, the $i$-th
  path secret is encrypted under node $v$'s keys.
  $L \leftarrow ()$
  $\mathsf{path} \leftarrow (\tau'.\mathsf{leafof}(\mathsf{id}_s)) \mathbin{+\!\!+} \tau'.\mathsf{directPath}(\tau'.\mathsf{leafof}(\mathsf{id}_s))$
  for $i = 1$ to len$(\mathsf{path}) - 1$ do
    $\vec{v} \leftarrow \mathsf{path}[i+1].\mathsf{resolution}() \setminus \mathsf{path}[i].\mathsf{resolution}()$
    for $j = 1$ to $|\vec{v}|$ do
      $L \mathbin{+\!\!\leftarrow} (i, \vec{v}[j])$
  return $L$

helper *weights$(\tau', \mathsf{id}_s)$
  // Returns a list of weights for $\mathsf{RPC}^{\mathrm{SAIK}}_{\ell,\vec{w}}$ when $\mathsf{id}_s$ commits in $\tau'$. Also allows to
  compute $\ell = |\vec{w}|$.
  $\vec{w} \leftarrow ()$
  $\mathsf{path} \leftarrow (\tau'.\mathsf{leafof}(\mathsf{id}_s)) \mathbin{+\!\!+} \tau'.\mathsf{directPath}(\tau'.\mathsf{leafof}(\mathsf{id}_s))$
  for $i = 2$ to len$(\mathsf{path})$ do
    $\vec{v} \leftarrow \mathsf{path}[i].\mathsf{resolution}() \setminus \mathsf{path}[i-1].\mathsf{resolution}()$
    for $j = 1$ to $|\vec{w}|$ do
      $\vec{w} \mathbin{+\!\!\leftarrow} \big|\{u \in \tau'.\mathsf{leaves} \mid u.\mathsf{inuse}() \wedge \tau'.\mathsf{isInSubtree}(u, \vec{v}[j])\}\big|$
  return $\vec{w}$
```

Fig. 17: The algorithms of SAIK.

SAIK: Creating epochs

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

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

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

helper *populate-secrets($\gamma'$, dk', $ctxt_1$, $ctxt_2$, id$_s$)
  try pathSec $\leftarrow$ mmPKE.Dec(dk, $ctxt_1$)
  try joinerSec $\leftarrow$ mmPKE.Dec(dk, $ctxt_2$)
  $v \leftarrow \gamma'.\tau$.lca($\gamma'$.leaf, $\gamma'.\tau$.leafof(id$_s$))
  while $v \neq \gamma'.\tau$.root do
    $r \leftarrow$ HKDF.Expand(pathSec, 'node')
    (ek, $v$.dk) $\leftarrow$ mmPKE.KG($r$)
    req $v$.ek = ek
    pathSec $\leftarrow$ HKDF.Expand(pathSec, 'path')
    $v \leftarrow v$.parent
  $\gamma' \leftarrow$ *derive-epoch-keys($\gamma'$, joinerSec)
  return $\gamma'$
```

SAIK: Generating keys for $\mathcal{F}_{\text{AKS}}$

```
helper *AKS-kgen()
  (ek, dk) $\leftarrow$ mmPKE.KG()
  (vk, sk) $\leftarrow$ RS.KeyGen()
  (ek', dk') $\leftarrow$ mmPKE.KG()
  return ((ek, vk, ek'), (dk, sk, dk'))
```

SAIK: Key schedule

```
helper *derive-keys($\gamma$, $\gamma'$, commitSec)
  joinerSec $\leftarrow$ HKDF.Extract($\gamma$.initSec, commitSec)
  $\gamma' \leftarrow$ *derive-epoch-keys($\gamma'$, joinerSec)
  return ($\gamma'$, joinerSec)

helper *derive-epoch-keys($\gamma'$, joinerSec)
  epSec $\leftarrow$ HKDF.Extract(joinerSec, $\gamma'$.groupCtxt())
  $\gamma'$.appSec $\leftarrow$ HKDF.Expand(epSec, 'app')
  $\gamma'$.membKey $\leftarrow$ HKDF.Expand(epSec, 'membership')
  $\gamma'$.initSec $\leftarrow$ HKDF.Expand(epSec, 'init')
  return $\gamma'$
```

SAIK: Tree hash

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

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

Fig. 18: Additional helper methods for SAIK.

E.6 Extraction Procedure for the Mailboxing Service

In this section, we describe how in practice the mailboxing service can compute SAIK messages delivered to parties. Recall that this is not formally part of our model, but it is a part of SAIK.

Recall that according SAIK, a party id_s performing operation act sends to the mailboxing service $(\mathsf{id}_s, \mathsf{act}, \mathit{Ctxt}, \mathsf{updEKs}, \mathsf{sig})$, where Ctxt is a multi-recipient ciphertext encrypting path secrets and updEKs is a list of new keys for id_s's path. In case of an add, id_s also sends $\mathsf{welcomeData}$ for the joiner.

When a receiver id_r wants to download its message, the service first sends id_s and act to id_r. Then, we have three cases:

id_r *is removed:* First, id_r sends to the service the reduction pattern $\mathsf{rp} = (\ell, 0, 1)$ it expects, id_s's verification key vk and grpId (to compute them, it executes the first 4 lines of Receive). The service computes the message $\vec{\mathsf{tbs}} = (\mathit{ctxt}_1, \ldots, \mathit{ctxt}_\ell) +\!\!+ ((\mathsf{id}_s, \mathsf{act}, \mathsf{grpId})) +\!\!+ \mathsf{updEKs}$ signed by id_s (Fig. 16 describes how id_s computes the signed value), where $\mathit{ctxt}_i = \mathsf{mmPKE.Ext}(\mathit{Ctxt}, i)$ for all i.[17] Then, it sends to id_r its signature $\mathsf{sig}' = \mathsf{Reduce}(\mathsf{vk}, \mathsf{sig}, \vec{\mathsf{tbv}}, \mathsf{rp})$.

id_r *joins:* The service sends $\mathsf{welcomeData}$ to id_r. Recall that id_r acts as two receivers of Ctxt: the i-th one for its path secret and the last, n-th one for its joiner secret. Therefore, id_r sends to the service i and n computed based on the ratchet tree τ in $\mathsf{welcomeData}$ and id_s. (In detail, id_r executes the helper method `*my-reduction-pattern`(τ, id_s) from Fig. 17 which outputs $(n, i, *)$.) The service sends back $\mathit{ctxt}_1 = \mathsf{mmPKE.Ext}(\mathit{Ctxt}, i)$ and $\mathit{ctxt}_2 = \mathsf{mmPKE.Ext}(\mathit{Ctxt}, n)$.

else: id_r sends to the service the reduction pattern $\mathsf{rp} = (\ell, i, j)$ it expects, id_s's verification key vk and grpId (to compute them, it executes the first 4 lines of Receive). The service computes $\vec{\mathsf{tbs}}$ as in the case where id_r is removed and sends to id_r its individual ciphertext $\mathit{ctxt} = \mathsf{mmPKE.Ext}(\mathit{Ctxt}, i)$, its public keys $\mathsf{updEKs}[1], \ldots, \mathsf{updEKs}[j]$ and signature $\mathsf{sig}' = \mathsf{Reduce}(\mathsf{vk}, \mathsf{sig}, \vec{\mathsf{tbs}}, \mathsf{rp})$.

An alternative solution. In the solution described above, receiving a message requires interaction between the delivery service and id_r. This is not a problem in typical scenarios, because they are both online at that moment. However, we note that there is an alternative solution which does not require interaction. Specifically, the messages sent by SAIK contain enough information for the service to compute the public part of the ratchet tree in any epoch. The tree, in turn, is sufficient to compute the message delivered to any party. The downside of this solution is that it requires the mailboxing service to store many ratchet trees (or many messages to re-compute them) for parties in different epochs.

F Proof of Theorem 4 (Security of SAIK)

Theorem 4. *Let $\mathcal{F}_{\text{CGKA}}$ be the CGKA functionality with predicates* **confidential** *and* **authentic** *defined in Fig. 7. Let SAIK be instantiated with schemes mmPKE, HRS and the HKDF functions modelled as a hash Hash. Let $\mathcal{A}$ be any environment. Denote the output of $\mathcal{A}$ from the real-world execution with SAIK and the hybrid setup functionality $\mathcal{F}_{\text{AKS}}$ from Fig. 14 as $\text{REAL}_{\mathsf{SAIK},\mathcal{F}_{\text{AKS}}}(\mathcal{A})$. Further, we denote the output of $\mathcal{A}$ from an ideal-world 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 $\mathcal{B}_5$ such that*

$$\begin{aligned}\Pr\left[\text{IDEAL}_{\mathcal{F}_{\text{CGKA}},\mathcal{S}}(\mathcal{A}) = 1\right] - \Pr\left[\text{REAL}_{\mathsf{SAIK},\mathcal{F}_{\text{AKS}}}(\mathcal{A}) = 1\right] \leq\ & \text{Adv}^{\mathsf{CR}}_{\mathsf{Hash}}(\mathcal{B}_1) \\ &+ q_e^2(q_e+1)\log(q_n) \cdot \text{Adv}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE},q_e \log(q_n),q_n}(\mathcal{B}_2) \\ &+ 3q_h q_e^2(q_e+1)/2^\kappa + 2q_e \cdot \text{Adv}^{\mathsf{AEUF\text{-}RCMA}}_{\mathsf{HRS}}(\mathcal{B}_3) \\ &+ 2q_e \cdot \text{Adv}^{\mathsf{RKC}}_{\mathsf{HRS}}(\mathcal{B}_4) + q_e \cdot \text{Adv}^{\mathsf{SEUF\text{-}RCMA}}_{\mathsf{HRS}}(\mathcal{B}_5),\end{aligned}$$

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.

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 $\mathcal{F}^1_{\text{CGKA}}$ which allows the simulator to set all outputs. This means that $\mathcal{F}^1_{\text{CGKA}}$ gives no security guarantees. 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

[17] In typical constructions, including ours, this is very efficient.

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.

F.1 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 8. *For any environment $\mathcal{A}$, there exists an adversary $\mathcal{B}$ such that*

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

where Hash *models the* HKDF.Expand *and* HKDF.Extract *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.

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 according to 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.

Proof. We next prove Theorem 8. 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 $\mathcal{A}$ triggers BreaksCons, then a reduction $\mathcal{B}$ can extract from a hash collision. (Theorem 8 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 the parties compute $\mathsf{epSec} \leftarrow \mathsf{Hash}(\mathsf{joinerSec}, \mathsf{groupCtxt})$ (Fig. 18), where $\mathsf{groupCtxt} = (\mathsf{grpId}, \mathsf{treeHash}, \mathsf{id}_s\text{-}\mathsf{act})$ (Table 4). 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 $\mathsf{treeHash}$. This implies agreement on the roster, which is encoded in the tree leaves.
b) The group key in epid which is derived as $\mathsf{Hash}(\mathsf{epSec}, \text{'}\mathit{app}\text{'})$.

Moreover, let epSec' denote the epoch secret of epid's parent. We have $\mathsf{joinerSec} = \mathsf{Hash}(\mathsf{initSec}', \mathsf{commitSec})$, where $\mathsf{initSec}' = \mathsf{Hash}(\mathsf{epSec}', \text{'}\mathit{init}\text{'})$ and $\mathsf{commitSec}$ 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 equation $\mathsf{joinerSec} = \mathsf{Hash}(\mathsf{initSec}', \mathsf{commitSec})$ 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 $\mathsf{BreaksCons}$ 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 $\mathsf{HG}[\mathsf{epid}].\mathsf{mem}$ 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 $\mathsf{HG}[\mathsf{epid}].\mathsf{mem}$. By agreement on act in a), id modifies the tree the same way as $\mathcal{F}^2_{\text{CGKA}}$ modifies $\mathsf{HG}[\mathsf{epid}].\mathsf{mem}$, which proves the statement.

F.2 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 $\mathsf{mmOW\text{-}RCCA}$ secure, then SAIK guarantees confidentiality, that is, that hybrids 2 and 3 are indistinguishable. Formally, we prove the following theorem.

Theorem 9. *For any environment $\mathcal{A}$, there exists an adversary $\mathcal{B}$ such that*

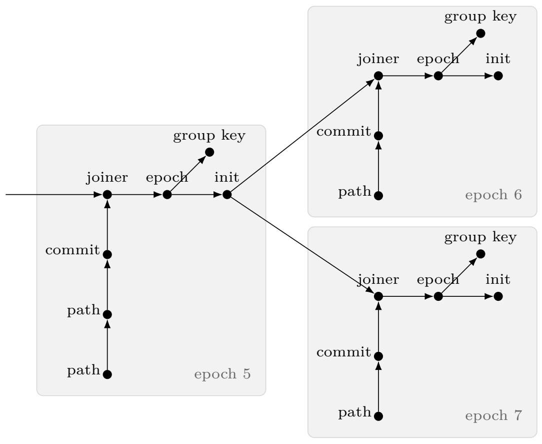
$$\Pr\left[\text{IDEAL}_{\mathcal{F}^3_{\text{CGKA}},\mathcal{S}^3}(\mathcal{A}) \Rightarrow 1\right] - \Pr\left[\text{IDEAL}_{\mathcal{F}^2_{\text{CGKA}},\mathcal{S}^2}(\mathcal{A}) \Rightarrow 1\right]$$
$$\leq 4q_e^2 q_h/2^\kappa + q_e^2 \log(q_n) \cdot \text{Adv}^{\mathsf{mmOW\text{-}RCCA}}_{\mathsf{mmPKE},q_e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

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 the RO.

Formally, it is left to prove the following lemma.

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

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

Bounding the probability of bad events. Recall that an epoch epid is confidential if `*grp-secs-secure`(epid) is true (all predicates are defined in Fig. 7). The latter predicate is recursive, starting at the root epoch with $\mathsf{epid} = 0$. Accordingly, we will prove a recursive upper bound on the probability of $\mathsf{SecsHashed}_{\mathsf{epid}}$. Formally, Lemma 4 is implied by the following lemma.

Lemma 5. *There exists a reduction $\mathcal{B}$ and some events $\mathsf{BreaksRCCA}_{\mathsf{epid}}$*[18] *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*,*

$$\Pr[\mathsf{SecsHashed}_{\mathsf{epid}}] \le 4q_h/2^\kappa + \Pr[\mathsf{SecsHashed}_{\mathsf{epid}_p}] + \Pr[\mathsf{BreaksRCCA}_{\mathsf{epid}}].$$

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 Lemma 5 implies Lemma 4, observe that Lemma 5 implies

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

where the first step follows from a) and b) in Lemma 5. Since by the union bound, $\Pr[\exists \mathsf{epid} : \mathsf{SecsHashed}_{\mathsf{epid}}] \le q_e \cdot \max_{\mathsf{epid}} \Pr[\mathsf{SecsHashed}_{\mathsf{epid}}]$, we get Lemma 4.

Proof of Lemma 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 $q_h/2^\kappa < 4q_h/2^\kappa$.

Proof of Lemma 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 $\mathsf{initSec}$, epSec, $\mathsf{joinerSec}$ and the rest of the experiment are as follows: 1) $\mathsf{initSec}$ is stored by parties in epid, 2) $\mathsf{joinerSec}$ is the output of the RO on input $\mathsf{commitSec}$ together with $\mathsf{initSec}_p$, 3) $\mathsf{joinerSec}$ 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 that either `*grp-secs-secure`(epid_p) 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, `*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, `*all-ind-secs-secure`(epid) is true and $\mathsf{commitSec}$ (of epid) is contained in some value inputted by $\mathcal{A}$ to the RO.

This means that unless $\mathsf{InitHashed}_{\mathsf{epid}_p}$ or $\mathsf{CommHashed}_{\mathsf{epid}}$ occurs, $\mathcal{A}$ has no information about $\mathsf{initSec}$ and epSec. Therefore, the best strategy for $\mathcal{A}$ to trigger $\mathsf{SecsHashed}_{\mathsf{epid}}$ is to either guess $\mathsf{initSec}$ 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 $\mathsf{commitSec}$ (of epid) together, but some value contains $\mathsf{joinerSec}$ (of epid).

Therefore, we have

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

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

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

Event $\mathsf{BreaksRCCA}_{\mathsf{epid}}$ **:** Either $\mathsf{CommHashed}_{\mathsf{epid}}$ or $\mathsf{JoinHashed}_{\mathsf{epid}}$ occurs.[19]

This proves the claim.

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

Lemma 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 $\mathsf{mmOW\text{-}RCCA}$ 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 $\mathsf{mmOW\text{-}RCCA}$ 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

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

□

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

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

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 $\mathsf{mmOW\text{-}RCCA}$ game as public keys

[19] Intuitively, the only dependency between the commit and joiner secrets comes from encryptions, so inputting the secrets to the RO requires breaking $\mathsf{IND\text{-}RCCA}$.

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).[20]

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 $\mathsf{mmOW\text{-}RCCA}$ 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.

<u>PRECISE DESCRIPTION OF $\mathcal{B}$.</u> 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 $\mathsf{mmOW\text{-}RCCA}$ 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 $\mathsf{mmOW\text{-}RCCA}$ 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 $\mathsf{mmOW\text{-}RCCA}$ 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 $\mathsf{mmOW\text{-}RCCA}$ 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.

[20] 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.

$\mathcal{B}$ 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 $\mathsf{mmOW\text{-}RCCA}$ 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 $\mathsf{mmOW\text{-}RCCA}$ 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 8. *If* `*all-ind-secs-secure`(epid) *is true, then for each v in τ, $v.\mathsf{ek}$ is generated during an honest send.*

Proof. Take any v in τ. Let epid_0 be the epoch which introduces $v.\mathsf{ek}$ 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`$(\mathsf{epid}, \mathsf{id}_s)$ is false and `*all-ind-secs-secure`(epid) is false, which contradicts $\mathsf{CommHashed}_{\mathsf{epid}}$. □

Let τ be the ratchet tree in epid. By Lemma 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 Lemma 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 $\mathsf{mmOW\text{-}RCCA}$ 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 Lemma 8, this means that v's key belongs to some $\mathsf{mmOW\text{-}RCCA}$ 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 9. *If* `*all-ind-secs-secure`(epid) *is true, then for each v in τ, no party holding $v.\mathsf{dk}$ is corrupted.*

Proof. Take any node v in τ. Further, let epid_0 be the epoch which introduces $v.\mathsf{ek}$ and let $\mathsf{epid}_1, \dots, \mathsf{epid}_\ell$ 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 $\mathsf{epid}_0, \dots, \mathsf{epid}_\ell$, because any modification replaces v's keys. Moreover, epid is one of these epochs, so this subtree is the same as in τ. Let $\mathsf{id}_1, \dots, \mathsf{id}_n$ be the parties in v's subtree in τ.

Second, we observe that if $\texttt{*all-ind-secs-secure}(\mathsf{epid})$ 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, which is one of $\mathsf{epid}_0, \dots, \mathsf{epid}_\ell$, 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 $\texttt{*ind-secs-secure}(\mathsf{epid}, \mathsf{id}_i)$ would be false and $\texttt{*all-ind-secs-secure}(\mathsf{epid})$ would be false, which contradicts $\mathsf{CommitHashed}_{\mathsf{epid}}$.

Finally, it is left to show that $v.\mathsf{dk}$ is held only by $\mathsf{id}_1, \dots, \mathsf{id}_n$ in epochs $\mathsf{epid}_0, \dots, \mathsf{epid}_\ell$. 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 $v^\perp.\mathsf{dk}$ 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 $v^\perp.\mathsf{dk}$ 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 $v^\perp.\mathsf{dk}$ into the state of $\mathsf{id}^\perp$. Further, let epid_0 be the epoch that introduces $v.\mathsf{dk}$ 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.[21] 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. □

F.3 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 HRS is unforgeable (SEUF-RCMA and AEUF-RCMA) and key committing (RKC) and if mmPKE is mmOW-RCCA secure, then SAIK guarantees authenticity, that is, hybrids 3 and 4 are indistinguishable. (Security of mmPKE is used to guarantee secrecy of the symmetric membership keys used by HRS.)

Game-based perspective. Observe that hybrids 3 and 4 are identical unless an authentic epoch is created by a message injected by $\mathcal{A}$. We call this bad event Forges. It is easy to see that $\mathcal{A}$'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.

[21] 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 $\mathcal{B}_{\mathsf{epid}}$ 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.

Two bad events. Let $\mathcal{A}$ be any environment. As stated above, hybrids 3 and 4 are identical unless $\mathcal{A}$ triggers Forges, so the goal is to upper bound the probability that it occurs. Since epochs in detached trees are not authentic and the root epoch with $\mathsf{epid} = 0$ cannot be injected by definition, in the remainder of the proof we only consider non-root epochs in the main history-graph tree. Such epochs are authentic in two cases: if in their parent epochs either the group secrets or the individual secrets of their creators are secure. Accordingly, we define two sub-events of Forges depending on which secrets are secure:

Event $\mathsf{ForgesAsym}$: There exists an epoch epid with parent epid_p and creator id_s such that $\mathsf{HG}[\mathsf{epid}].\mathsf{inj}$ and `*ind-secs-secure`$(\mathsf{epid}_p, \mathsf{id}_s)$ are both true.

Event $\mathsf{ForgesSym}$: There exists an epoch epid with parent epid_p such that $\mathsf{HG}[\mathsf{epid}].\mathsf{inj}$ and `*grp-secs-secure`(epid_p) are both true.

It remains to bound the probability of each $\mathsf{ForgesAsym}$ and $\mathsf{ForgesSym}$.

Notation. In case $\mathsf{ForgesAsym}$ or $\mathsf{ForgesSym}$ occurs for epoch epid with parent epid_p, we denote by id_s the creator of epid. Further, we let id_r be the group member accepting the injected message creating epid and we let vk_r, k_r, $\vec{m}_r$, rp_r and σ_r be the values inputted by id_r to $\mathsf{HRS.Vrfy}$ when accepting the message.

Asymmetric forgery. We next prove the following lemma

Lemma 10. *There exist reductions $\mathcal{B}_1$ and $\mathcal{B}_2$ such that*

$$\Pr[\mathsf{ForgesAsym}] \leq 2q_e \cdot \mathrm{Adv}^{\mathsf{AEUF\text{-}RCMA}}_{\mathsf{HRS}}(\mathcal{B}_1) + 2q_e \cdot \mathrm{Adv}^{\mathsf{RKC}}_{\mathsf{HRS}}(\mathcal{B}_2).$$

Both $\mathcal{B}_1$ and $\mathcal{B}_2$ run $\mathcal{A}$ identically; the only difference between them is in how they find forgeries. Specifically, the reductions emulate hybrid 3 for $\mathcal{A}$ and embed 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. Which key is replaced by vk^* is chosen at random. The reductions use the Sign oracle to sign honestly sent messages that verify with vk^*. If a party holding the corresponding sk^* is corrupted, they give up.

The next three claims show that with probability at least $\Pr[\mathsf{ForgesAsym}]/(2q_e)$, both $\mathsf{ForgesAsym}$ occurs and $\mathsf{vk}_r = \mathsf{vk}^*$ (see notation above). Moreover, if this happens, then one of the reductions wins.

Claim. The probability of $\mathsf{vk}_r = \mathsf{vk}^*$ is at least $\Pr[\mathsf{ForgesAsym}]/(2q_e)$

Proof. We will show that if $\mathsf{ForgesAsym}$ occurs, then vk_r is honestly generated. Since there are at most $2q_e$ honestly generated keys, this proves the claim.

Assume $\mathsf{ForgesAsym}$ 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`$(\mathsf{epid}_0, \mathsf{id}_s)$ would be true and `*ind-secs-secure`$(\mathsf{epid}_s, \mathsf{id}_s)$ would be false, which contradicts $\mathsf{ForgesAsym}$.

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. $\square$

Both reductions lose in case a party holding sk^* is corrupted. We next show that under right conditions this does not happen.

Claim. If $\mathsf{ForgesAsym}$ occurs and $\mathsf{vk}_r = \mathsf{vk}^*$, then no party holding sk^* is corrupted.

Proof. Assume towards a contradiction that $\mathsf{ForgesAsym}$ occurs, $\mathsf{vk} = \mathsf{vk}^*$ and a party holding sk^* is corrupted in some epoch $\mathsf{epid}^{\perp}$. Let $\mathsf{epid}_0^{\perp}$ be the epoch before $\mathsf{epid}^{\perp}$ which introduces sk^* 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^* into the state of id_r. If, on the other hand, $\mathsf{epid}_0^{\perp}$ 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 all of which were not created by id_s and not removing it, as this would remove sk^*. 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 `*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 `*exposed-ind-secs-weak`$(\mathsf{epid}_s, \mathsf{id}_s)$ would be true, which again makes the predicate `*ind-secs-secure`$(\mathsf{epid}_s, \mathsf{id}_s)$ false. □

It is left to explain how $\mathcal{B}_1$ or $\mathcal{B}_2$ can find a forgery with which it wins.

Claim. If ForgesAsym occurs and $\mathsf{vk}_r = \mathsf{vk}^*$, then either $\mathcal{B}_1$ wins with the forgery k_r, $\vec{m}_r$, rp_r and σ_r, or there are k' and $\vec{m}'$ such that $\mathcal{B}_2$ wins with k', k_r, $\vec{m}'$, rp_r and σ_r.

Proof. Assume ForgesAsym occurs and $\mathsf{vk} = \mathsf{vk}^*$. Further, assume that $\mathcal{B}_1$ does not win with k_r, $\vec{m}_r$, rp_r and σ_r. This can only happen if the **req** statement in the AEUF-RCMA game is violated, i.e., there exists an $\vec{m}'$ sent to the Sign oracle such that $\vec{m}_r = \mathsf{rp}_r(\vec{m}')$. $\mathcal{B}_1$ only sends $\vec{m}'$ to the Sign oracle when id_s creates epochs (since this is the only party signing with the honest sk^*). Let epid' denote the epoch created when $\vec{m}'$ was sent to Sign, and let epid'_p denote the parent of epid'.

If $\mathsf{epid}_p \neq \mathsf{epid}'_p$, then $k \neq k'$, because membership keys k and k' are derived from epoch keys, which uniquely identify epochs. Therefore, in this case $\mathcal{B}_2$ wins.

Finally, we show that $\mathsf{epid}_p = \mathsf{epid}'_p$ leads to a contradiction. Observe that in this case, the group state in epid' is computed by id_s who modifies the state in epid_p consistently with $\vec{m}'$. Further, the group state in epid is computed by id_r who also modifies the state in epid_p, this time consistently with $\vec{m}_r$ and rp_r. Since $\vec{m}_r = \mathsf{rp}_r(\vec{m}')$, these modifications are the same. Therefore, $\mathsf{epid} = \mathsf{epid}'$. However, epid is injected and epid' is honest, which is a contradiction. □

Symmetric forgery. We next prove the following lemma

Lemma 11. *There exist reductions $\mathcal{B}_3$ and $\mathcal{B}_4$ such that*

$$\Pr[\mathsf{ForgesSym}] \leq q_e \cdot \mathrm{Adv}_{\mathsf{HRS}}^{\mathsf{SEUF\text{-}RCMA}}(\mathcal{B}_3) + 3q_e^2 q_h / 2^\kappa + q_e^3 \log(q_n) \cdot \mathrm{Adv}_{\mathsf{mmPKE}, q_e \log(q_n), q_n}^{\mathsf{mmOW\text{-}RCCA}}(\mathcal{B}_4).$$

At a high level, $\mathcal{B}_3$ first guesses the epochs epid and epid_p that make ForgesSym occur. Then, it emulates hybrid 3 for $\mathcal{A}$, except instead of the membership key in epid_p, $\mathcal{B}_3$ uses its SEUF-RCMA Sign and Verify oracles. The fact that `*grp-secs-secure`(epid_p) is true will ensure that if $\mathcal{A}$ distinguishes $\mathcal{B}_3$'s emulation from hybrid 3, then a reduction $\mathcal{B}_4$ wins the mmOW-RCCA game. Further, the fact that epid is injected will ensure that $\mathcal{B}_3$ wins. A formal proof follows.

Consider the following experiments.

- **Hybrid 3.1 :** The same as hybrid 3, except at the beginning $\mathcal{A}$ announces an epoch epid_p and the experiment stops as soon as **confidential**(epid_p) becomes false.
- **Hybrid 3.2 :** The same as hybrid 3.1, except the membership key membKey in epid_p announced by $\mathcal{A}$ is random and independent.

Further, we define events analogous to ForgesSym but in hybrids 3.1 and 3.2:

- **Events $\mathsf{ForgesSym}_i$ for $i \in \{1, 2\}$:** At the end of hybrid i, there exists an epoch epid with parent epid_p announced by $\mathcal{A}$ such that $\mathsf{HG}[\mathsf{epid}].\mathsf{inj}$ is true.

Clearly, for the environment $\mathcal{A}'$ that guesses epid_p, we have $\Pr[\mathsf{ForgesSym}_1] \geq 1/q_e \cdot \Pr[\mathsf{ForgesSym}]$. It remains to upper bound $\Pr[\mathsf{ForgesSym}_1]$. We do this in two steps, formalized by the next two claims.

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

$$\Pr(\mathsf{ForgesSym}_2) - \Pr(\mathsf{ForgesSym}_1) \leq q_e^2 \log(q_n) \cdot \mathrm{Adv}_{\mathsf{mmPKE}, q_e, q_e \log(q_n), q_n}^{\mathsf{mmOW\text{-}RCCA}}(\mathcal{B}_2) + 3q_e^2 q_n / 2^\kappa.$$

Proof. Observe that the membership key is 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 is analogous to the proof of Theorem 9. □

Claim. There exists a reduction $\mathcal{B}_1$ such that

$$\Pr(\mathsf{ForgesSym}_2) \leq \mathrm{Adv}^{\mathsf{SEUF\text{-}RCMA}}_{\mathsf{HRS}}(\mathcal{B}_1).$$

Proof. $\mathcal{B}_1$ emulates hybrid 3.2 for $\mathcal{A}'$, except instead of the membership key in epid_p announced by $\mathcal{A}'$, $\mathcal{B}_1$ uses its $\mathsf{SEUF\text{-}RCMA}$ Sign and Verify oracles. Observe that **confidential**(epid_s) is always true, so no party is corrupted in epid_p. Membership keys are unique per epochs, so this means that the key from $\mathcal{B}_1$'s game is not leaked upon corruption. Therefore, $\mathcal{B}_1$ simulates the experiment perfectly.

If $\mathsf{ForgesSym}_2$ occurs for a child epid of epid_p, $\mathcal{B}_1$ halts and sends to the challenger vk, $\vec{m}$, rp and σ inputted by an id_r who transitioned to epid from epid_p as a current group member (i.e., id_r does not join into epid). Note that there must be such a member, because epid_p is injected, so it is created when an id_r receives a message and it is not in a detached tree, so it must have been attached when some id_r transitioned there from the main tree.

Assume towards a contradiction that $\mathcal{B}_1$ does not win with the above solution. Since id_r checked that the HRS verification outputs 1, this can only happen if $\mathcal{B}_1$ sent to the Sign oracle a $\vec{m}'$ such that $\vec{m} = \mathsf{rp}(\vec{m}')$. $\mathcal{B}_1$ uses the oracle only for messages sent by parties in epid_p (since membership keys are unique). Therefore, $\vec{m}'$ was sent to the oracle when some id_s created an epoch epid' as a child of epid_p. Observe that the group state in epid' is computed by id_s who modifies the state in epid_p consistently with $\vec{m}'$. Further, the group state in epid is computed by id_r who also modifies the state in epid_p, this time consistently with $\vec{m}_r$ and rp_r. Since $\vec{m}_r = \mathsf{rp}_r(\vec{m}')$, these modifications are the same. Therefore, $\mathsf{epid} = \mathsf{epid}'$. However, epid is injected and epid' is honest, which is a contradiction. $\square$