Server-Aided Continuous Group Key Agreement

Joël Alwen
alwenjo@amazon.com
AWS Wickr
New York, NY, USA

Dominik Hartmann
dominik.hartmann@rub.de
Ruhr University Bochum
Bochum, Germany

Eike Kiltz
eike.kiltz@rub.de
Ruhr University Bochum
Bochum, Germany

Marta Mularczyk
mulmarta@amazon.ch
AWS Wickr
New York, NY, USA

ABSTRACT

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

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

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

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

CCS CONCEPTS

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

CCS '22, November 7–11, 2022, Los Angeles, CA, USA

ACM ISBN 978-1-4503-9450-5/22/11.
https://doi.org/10.1145/3548606.3560632

KEYWORDS

secure group messaging; CGKA; MLS; E2E encryption

ACM Reference Format:
Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. 2022. Server-Aided Continuous Group Key Agreement. In *Proceedings of the 2022 ACM SIGSAC Conference on Computer and Communications Security (CCS '22), November 7–11, 2022, Los Angeles, CA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3548606.3560632

1 INTRODUCTION

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

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

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

Consequently, a new generation of E2E protocols is being developed both in academia (e.g. [1, 3–5, 7, 9, 23, 28]) and industry [13].

Their primary design goal is to support extremely large groups (e.g. 10s of thousands of users) while still meeting, or exceeding, the security and functionality of today's state-of-the-art deployed E2E protocols. Technically, the new protocols do this by reducing their communication complexity to *logarithmic* in the group size; albeit, only under favorable conditions in the execution. This informal property is sometimes termed the *fair-weather complexity*.

To date, the most important of these new E2E protocols is the IETF's upcoming secure group messaging (SGM) standard called the *Messaging Layer Security* (MLS) protocol. MLS is in the final stages of standardization and its core components are already seeing initial deployment [22].

Continuous Group Key Agreement. To the best of our knowledge, all next gen. E2E protocols share the following basic design paradigm. At their core lies a *Continuous Group Key Agreement* (CGKA) protocol; a generalization to the group setting of the *Continuous Key Agreement* 2-party primitive [4, 25] underlying the Double Ratchet.

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

MLS too, is (implicitly) based on a CGKA, originally dubbed *TreeKEM* [16]. Since its inception, TreeKEM has undergone several substantial revisions [10, 11] before reaching its current form [9, 12]. For clarity, we refer to its current version at the time of this writing as *Insider-Secure TreeKEM* (ITK) (using the terminology of [9] where that version was analyzed). ITK has already seen its first real world deployment as a core component of Cisco's Webex conferencing protocol [22].

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

1.1 Our Contributions

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

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

Security for CGKA. We define a new security notion for (sa)CGKA capturing the same intuitive guarantees as those shown for ITK [9] for example. Like other notions based on the history graph paradigm of [6], our notion is parameterized by *safety* predicates that together decide the security of a given epoch key in a given execution.

However, at a technical level our notion departs significantly from past ones. Essentially, it relaxes the requirement that group members in an epoch agree on and authenticate the *history of network traffic* leading to the epoch. Instead, the new notion "only" ensures they agree on and authenticate the *semantics* of the history; i.e. the "meaning" of the traffic rather than exact packet contents. This has several interesting consequences. First, it more directly captures our intuitive security goals. E.g. it avoids subtle questions about what intuition is really captured when, say, an AEAD ciphertext in a packet can be decrypted to different plaintexts using different keys.[1] Second, the relaxation creates wiggle room we can use to prove security despite group members no longer having the same view of network traffic. Finally, it allows us to relax the security of the encryption scheme used in our construction from CCA to *replayable CCA* (RCCA) [20].[2]

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

[1] This can happen for widely used AEADs like AES-GCM [24].
[2] This makes sense as RCCA was designed to relax the "syntactic non-malleability" of CCA to a form of "semantic non-malleability".

notion can easily be extended to a "full-fledged" (sa)CGKA using standard techniques.

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

Multi-message multi-recipient PKE.. First, we replace ITK's use of standard (CCA secure) PKE by multi-message multi-recipient PKE (mmPKE) [32]. mmPKE has the functionality of a parallel composition of standard PKE schemes (both in terms of ciphertext sizes and computation cost of encryption). Constructing mmPKE directly can result in a significantly more efficient scheme.

We introduce a new security notion for mmPKE, more aligned with the needs of (the security targeted by) SAIK. It both strengthens and weakens past notions: On the one hand, proving SAIK secure demands that we equip the mmPKE adversary of [32] with adaptive key compromise capabilities. On the other hand, thanks to the relaxation to semantic agreement, we "only" require RCCA security rather than full-blown CCA used in previous' works [7, 9].

We prove that the mmPKE construction of [32] satisfies our new notion based on a form of gap Diffie-Hellman assumption, the same as in [32]. The reduction is tight in that the security loss is independent of the number of parties (i.e. key pairs) in the execution (although it does depend on the number of corrupted key pairs). Moreover, we extend the proof to capture mmPKE constructions based on "nominal groups" [2]. Nominal groups abstract the algebraic structure over bit-strings implicit to the X25519 and X448 scalar multiplication functions and corresponding twisted Edwards curves.[31]. In practical terms, this means our proofs also apply to instantiations of [32] based on the X25519 and X448 functions.

Authentication. Second, we modify the mechanisms used by ITK to ensure members transitioning to a new epoch authenticate the sender announcing a new epoch. Rather than sign the full packet like in ITK, the sender in SAIK only signs a small tag which "binds" all salient properties of the new epoch, i.e., its secrets, the set of group members, the history of applied operations, etc. In fact, we use a tag that already exists in ITK (called the "confirmation tag").

Performance Evaluation. Finally, we compare the communication complexity of SAIK, ITK and the CGKA of [28] called CmPKE. We break down the communication cost into sender and receiver bandwidth, i.e., the size of a packet uploaded, resp., downloaded (by one receiver) from the server. This metric reflects the resources needed from an individual client.

We note that the sender bandwidth of CmPKE grows linearly in the group size, while for SAIK and ITK it varies depending on both the group size and the history of preceding operations. Meanwhile, the receiver bandwidth is independent of both the size and the history for CmPKE, grows logarithmically in the group size for SAIK and varies with both the size and history for ITK.

We find that compared to ITK, SAIK always requires less bandwidth (regardless of history and group size). However, compared to CmPKE, SAIK requires slightly more receiver bandwidth but anywhere from the same to far less sender bandwidth. Concretely, in a group with 10K parties, CmPKE's sender bandwidth is 0.8MB while SAIK and ITK's bandwidths range between 3.6KB - 0.8MB and 4.4KB - 1.5MB, respectively (depending on the history). Meanwhile for receivers CmPKE and SAIK require 0.8KB and 2KB respectively while ITK requires between 4.4KB - 1.5*MB*.

In addition, we also compare the total bandwidth considered in [28], i.e., the size of the uploaded packet and all downloaded packets together. This metric reflects the resources required from the server, or equivalently from all clients together. We find that SAIK requires more total bandwidth than CmPKE but *much* less than ITK.

Outline. The paper is structured as follows. Sec. 2 covers preliminaries. Sec. 3 focuses on mmPKE. Sec. 4 describes the new security model for saCGKA with details outsourced to the full version [8]. Sec. 5 describes the SAIK protocol with details found in [8]. Sec. 6 formally states SAIK's security. Sec. 7 contains empirical evaluation and comparison of SAIK to previous constructions. Finally, Sec. 8 contains extensions to stronger security guarantees. SAIK's formal security proof is formalized in the full version [8].

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 [23]. This was soon followed by the first version of TreeKEM [33] which evolved to add stronger security [10, 33, 35] and more flexible functionality [11] culminating in its current form ITK [9] reflected in the current draft of the MLS RFC [12].

Reducing the communication complexity of TreeKEM and its descendants is not a new goal. *Tainted TreeKEM* [3] exhibits an alternate complexity profile optimized for a setting where the group is managed by a small set of moderators. Recently, [1] introduced new techniques for 'multi-group" CGKAs (i.e. CGKAs that explicitly accommodate multiple, possibly intersecting, groups) with better complexity than obtained by running a "single-group" CGKA for each group. Other work has focused on stronger security notions for CGKA both in theory [7] and with an eye towards practice [5, 9]. Supporting more concurrency has also been a topic of focus as witnessed by the protocols in [11, 18, 36]. Recently [27] present CGKA with novel membership hiding properties.

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

does not capture adversaries that deliver packets with ciphertexts for which they do not know the plaintexts.

Indeed, a good indication that such simplifications can be problematic can be found in [9]. They present an attack on TreeKEM (that can easily be easily adapted to the CGKAs in the above works except for [19]) which uses honest group members as decryption oracles to clearly violate the intuitive security expected of a CGKA. Yet, each of the above works (except for [19]) proves security of their CGKA using only IND-CPA secure encryption.

In contrast to the above works, [7] aimed to capture the full capabilities a realistic adversary might have. Thus, they model a fully active adversary that can leak parties local states at will and even set their random coins. In [9] this setting is extended to capture *insider* security. That is adversaries which can additionally corrupt the PKI. This captures the standard design criterion for deployed E2E applications that key servers are *not* considered trusted third parties. Unfortunately, this level of real-world accuracy has resulted in a (probably somewhat inherently) complicated model.

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

The CGKA of [28]. The work [28] presents a variant of CGKA, called here *filtered* CGKA (fCGKA), along with a protocol called CmPKE. In fCGKA, like in saCGKA, receivers download personalized sub-packets. However, fCGKA achieves this differently — an uploaded fCGKA packet has a particular form, namely, a header delivered to all receivers, followed by a number of ciphertexts, one for each receiver. Note that fCGKA is a special case of saCGKA where the Extract procedure outputs the header and the receiver's ciphertext.

The fCGKA security notion in [28] is essentially the model of [9]. The only difference is that [9] requires agreement on the history of the network packets leading to a given epoch. To adapt this to the fCGKA syntax, [28] requires instead agreement on the history of *packet headers*. Compared to our saCGKA notion, this is still syntactic agreement and e.g., requires CCA security. See Sec. 4.5.

Regarding the communication cost, CmPKE is designed to reduce the *total bandwidth*, i.e., for an operation, it minimizes the size of the sent packet and all downloaded packets together. In contrast, SAIK is designed to reduce the *maximum bandwidth*, i.e., it minimizes the size of each sent or downloaded packet. Accordingly, CmPKE has smaller total bandwidth than SAIK. In fact, the maximum size of a downloaded packet is also smaller for CmPKE. However, the size of a sent packet is usually much bigger for CmPKE. In summary, SAIK is designed to support clients with poor bandwidth, i.e., it minimizes the size of a single uploaded or downloaded packet. Thus, while the server load is a bit higher, the network requirements for clients are usually much lower.

mmPKE.. mmPKE was introduced by Kurosawa [30] though their security model was flawed as pointed out and fixed by Bellare et.al [14, 15]. Yet, those works too lacked generality as they demanded malicious receivers know a secret key for their public key. This restriction was lifted by Poettering et.al. in [32] who show that well-known PKE schemes such as ElGamal[26] are secure even when reusing coins across ciphertexts. Indeed, reusing coins this way can also reduce the computational complexity of encapsulation and the size of ciphertexts for KEMs as shown in the Multi-Recipient KEM (mKEM) of [21, 29, 34] 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 only [28] allows for adaptive corruption of honest keys, which is necessary for ITK's security against adaptive adversaries.

2 NOTATION

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

3 MULTI-MESSAGE MULTI-RECIPIENT PKE

We first recall the syntax of mmPKE from [15]. At a high level, mmPKE is standard encryption that supports batching a number of encryption operations together, in order to improve efficiency.[3]

*Definition 3.1 (*mmPKE*).* A Multi-Message Multi-Receiver Public Key Encryption (mmPKE) 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 key pair.
- $\mathsf{Enc}(\vec{\mathsf{ek}}, \vec{m}) \xrightarrow{\$} C$: On input of a vector of public keys $\vec{\mathsf{ek}}$ and a vector of messages $\vec{m}$ of the same length, outputs a *multi-recipient ciphertext* C encrypting each message in $\vec{m}$ to the corresponding key in $\vec{\mathsf{ek}}$.
- $\mathsf{Ext}(C, i) \to c_i$: A deterministic function. On input of a multi-recipient ciphertext C and a position index i, outputs an *individual ciphertext* c_i for the i-th recipient.
- $\mathsf{Dec}(\mathsf{dk}, c) \to m \vee \perp$: On input of an individual ciphertext c and a secret key dk, outputs either the decrypted message m or, in case decryption fails, $\perp$.

3.1 Security with Adaptive Corruptions

Our security notion for mmPKE, called mmIND-RCCA, 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 [32], but there are two important differences: First, [32] does not consider corruptions. Second, instead of CCA, we define the slightly weaker notion of Replayable CCA (RCCA). Roughly, RCCA [20] is the same

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

as CCA except modifying a ciphertext so that it encrypts the exact same message is not considered an attack. RCCA security is implied by CCA security.

We note that an almost identical security definition was presented in parallel by Hashimoto et.al.[28]. However, they only consider multi-recipient PKE (mPKE), where all recipients receive the same message.

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

The last requirement means that a secure mmPKE scheme may leak the lengths of components of encrypted vectors. We note that SAIK is also secure when instantiated with an mmPKE scheme that leaks more (see full version [8]), e.g. whether two messages in a vector are the same. The formal definitions are in the full version [8].

3.2 Construction

The mmPKE of [32] is straightforward. It requires a data encapsulation scheme DEM, a hash H and a group $\mathbb{G}$ of prime order p generated by g.[4] Recall that ElGamal encryption of m to public key g^x requires sampling coins r to obtain ciphertext $(g^r, \mathsf{DEM}(k_m, m))$ where $k_m = H(g^{rx}, g^x)$. The mmPKE variant reuses coins r from the first ElGamal ciphertext to encrypt all subsequent plaintexts. Thus, the final ciphertext has the form $(g^r, \mathsf{DEM}(k_1, m_1), \mathsf{DEM}(k_2, m_2), \ldots)$ where $k_i = H(g^{rx_i}, g^{x_i})$ for all i. We call the construction $\mathsf{DH\text{-}mmPKE}[\mathbb{G}, g, p, \mathsf{DEM}, H]$; a formal description is in the full version [8].

Optimizing for Short Messages. Normally, when messages m can have arbitrary size, a sensible mmPKE would use a KEM\DEM style construction to avoid having to re-encrypt m multiple times. In other words, for each m in the encrypted vector, we choose a fresh key k'_m for an AEAD and encrypt m with k'_m. Then use the mmPKE of [32] to encrypt k'_m to each public key receiving m. However, since the secrets encrypted in SAIK have the same length as AEAD keys, in our case it is more efficient to encrypt the secrets directly.

Tight security bound. In the full version [8], we prove the following upper bound on the advantage of any adversary against the mmPKE from [32]. Our bound is tighter than the bound that follows from the straightforward adaptation of the bound from [32] (i.e. using the hybrid argument and guessing the uncorrupted key). In particular, that bound would depend (linearly) on the total number of public keys, which may get very large. In contrast, our bound depends only on the number of corrupted keys and the length of encrypted vectors.

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

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

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

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

Remark 1. Some practical applications of Diffie-Hellman, most notably Curve25519 and Curve448 [31], implement a Diffie-Hellman operation that is not exponentiation in a prime-order group. Such operations can be formalized as so-called nominal groups [2]. In the full version [8], we generalize and prove Theorem 3.2 for nominal groups. In particular, this means that DH-mmPKE is secure if instantiated with Curve25519 and Curve448.

4 SERVER-AIDED CGKA

In this section, we first explain the saCGKA syntax. Then, we give intuitive security properties saCGKA protocols should provide and an overview of our saCGKA security model. For details, see the full version [8]. Finally, we highlight the additional flexibility provided by semantic agreement of saCGKA and list simplifications it makes compared with previous works on active CGKA security [7, 9, 28].

4.1 Syntax

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

4.2 Intuitive Security Properties

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

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

In a perfect execution, the history graph would be a chain. However, even for benign reasons, this may not be the case. For example,

if two parties simultaneously create epochs, then a fork in the graph is created. Moreover, an active adversary can deliver different messages to different parties, causing them to follow different branches. Further, it can trick parties into joining fake groups it created by injecting invitation messages. Epochs in fake groups form what we call *detached trees*. So, in full generality, the graph is a directed forest. Using history graphs we can list intuitive security properties of saCGKA.

Consistency Any two parties in the same epoch E 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 history: the parties reached E by executing the same sequence of group modifications since the latter one joined.

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

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

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

4.3 Authenticated Key Service (AKS)

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

4.4 Formal Model Intuition

We define security of saCGKA protocols in the UC framework. That is, a saCGKA protocol is secure if no environment $\mathcal{A}$ can distinguish between the real world where it interacts with parties executing the protocol and the ideal world where it interacts with the ideal saCGKA functionality and a simulator. Readers familiar with game-based security should think of $\mathcal{A}$ as the adversary (see also [8] for some additional discussion).

The real world. In the real-world experiment, the following actions are available to $\mathcal{A}$: First, it can instruct parties to perform different group operations, creating new epochs. When this happens, the party runs the protocol, updates its state and hands to $\mathcal{A}$ the message meant to be sent to the mailboxing service. The mailboxing service is fully controlled by $\mathcal{A}$. This means that the next action it can perform is to deliver arbitrary messages to parties. A party receiving such a message updates its state (or creates it in case of new members) and hands to $\mathcal{A}$ the semantic of the group operation it applied. Moreover, $\mathcal{A}$ can fetch from parties group keys computed according to their current states and corrupt them by exposing their current states.[5]

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

Security guarantees in the ideal world. To formalize confidentiality, $\mathcal{F}_{\text{CGKA}}$ is parameterized by a predicate **confidential**, which determines the epochs in the history graph in which confidentiality of the group key is guaranteed. For such a confidential epoch, $\mathcal{F}_{\text{CGKA}}$ chooses a random and independent group key. Otherwise, the simulator chooses an arbitrary key. To formalize authenticity, $\mathcal{F}_{\text{CGKA}}$ is parameterized by **authentic**, which determines if authenticity is guaranteed for an epoch and a party. As soon as an injected epoch with authentic parent appears in the history graph, $\mathcal{F}_{\text{CGKA}}$ halts, making the worlds easily distinguishable. Finally, $\mathcal{F}_{\text{CGKA}}$ guarantees consistency by computing the outputs, such as the set of group members outputted by a joining party, based on the history graph. This means that the outputs in the real world must be consistent with the graph (and hence also with each other) as well, else, the worlds would be distinguishable.

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

Personalizing messages. saCGKA protocols may require that the mailboxing service personalizes messages before delivering them. In our model, such processing is done by $\mathcal{A}$. It can deliver an honestly processed message, or an arbitrary injected message. The simulator decides if a message is honestly processed, i.e., leads to a non-injected epoch, or is injected, i.e., leads to an injected epoch.

[5] 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.

Note that this notion has an RCCA flavor. For example, delivering an otherwise honestly generated message but with some semantically insignificant bits modified can lead the receiver to an honest epoch.

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

We note that, in general, there can exist protocols that achieve the following stronger guarantee: Upon a trivial-win corruption, $\mathcal{F}_{\text{CGKA}}$ gives to the simulator the random key it chose and the simulator comes up with a fake state that matches it. However, this requires techniques which typically are expensive and/or require additional assumptions, such as a random oracle programmable by the simulator or a common-reference string. We note that the disadvantage of the simpler weaker 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. Our ideal world is agnostic to the protocol, which is conceptually simple.

4.5 Semantic Agreement

An important difference between our model and those of [7, 9, 28] is that in [7, 9, 28] epochs are (uniquely) identified by messages creating them. This is problematic for saCGKA, because different receivers transition to a given epoch using different messages. Crucially, this means an injected message cannot be used to identify the injected epoch into which its receiver transitions. We deal with this in a clean way by allowing the simulator to identify epochs. That is, epoch identifiers are arbitrary as long as consistency, authenticity and confidentiality hold.

The work [28], which proposes a new CGKA where, similar to SAIK, receivers get personalized packets, encountered the same problem with the existing models [7, 9]. In their new model, filtered CGKA (fCGKA), an epoch is identified by the sequence of *packet headers* leading to it. The header is a part of the uploaded packet that is downloaded by all receivers. A protocol can be secure according to the fCGKA model only if the header it defines has the properties of a cryptographic commitment to the semantics of the packet.

saCGKA generalizes fCGKA (and [7, 9]) and provides additional flexibility. For instance, it enables CGKAs which, like SAIK, assume PKE with the weaker RCCA security, while fCGKA still requires the stronger notion of CCA. We believe that in the future more CGKA protocols will take advantage of saCGKA's flexibility. For example, one may consider using a different packet-authenticator for each receiver with the goal of providing some level of unlinkability – an adversary seeing only packets downloaded by participants cannot tell if they are in the same epoch (or group) or not.

4.6 Simplifications

In order to make the security notion tractable, we made the following simplifications compared to the models of [7, 9, 28].

Immediate transition In our model, a party performing a group operation immediately transitions to the created epoch. In reality, a party would only send the message creating the epoch and wait for an ACK from the mailboxing service before transitioning. If it receives a different message before the ACK, it transitions to that epoch instead. This mitigates the problem that if many parties send at once then they end up in parallel epochs and cannot communicate.
A protocol *Prot* implementing immediate transition can be transformed in a black-box manner into a protocol *Prot'* that waits for ACK as follows: To perform a group operation, *Prot'* creates a copy of the current state of *Prot* and runs *Prot* to obtain the provisional updated state and the message. The message is sent and all provisional states are kept in a list. If some message is ACK'ed, the corresponding provisional state becomes the current one, and if another message is received, it is processed using the current state. In any case, all provisional states are cleared upon transition.

Simplified PKI The models of [9, 28] consider a realistic implementation of the AKS where parties generate key packages themselves and upload them to an untrusted server, authenticated with long-term so-called identity keys. These long-term keys are authenticated via a PKI which allows the adversary to leak registered keys and even to register their own arbitrary keys on behalf of any participant. The works [9, 28] define fine-grained security in this setting, i.e., their security predicates take into account which PKI keys delivered to parties were corrupted.
In contrast, our model avoids the complexity of keeping track of the PKI keys in $\mathcal{F}_{\text{CGKA}}$, at the cost of more coarse-grained guarantees. For example, it no longer captures the (subtle) security guarantees provided by (the tree-signing of) ITK to parties invited to fake groups created by the adversary (tree signing trivially works for SAIK). We stress that our

model does capture most active attacks, e.g. injecting valid-looking packets that add parties with arbitrary injected key packages.

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

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

One can easily adapt the special-casing of [7, 9, 28] to our model. We chose not to do this for simplicity and because well-designed protocols, including ITK and SAIK, naturally have protections against bad randomness.(Looking ahead, these protocols mix a fresh "commit" secret for the new epoch with the "init" secret from the old epoch, which mitigates sampling the fresh secret with bad randomness.) Therefore, capturing the additional attack vector typically does not fundamentally improve the analysis.

Simplified syntax To improve efficiency, ITK and CmPKE of [28] use the so-called propose-commit syntax, originally proposed by MLS. This means that parties first send messages that *propose* adding or removing other members, or updating their own keys. This does not affect the group state immediately. Rather, a party can collect a list of proposals and send a *commit* message which applies the proposed changes and creates a new epoch. The advantage of this is avoiding the expensive operation of epoch creation after every group modification (modifications typically come in batches; for instance, lots of members are added immediately after group creation).

Unfortunately, using this syntax requires lots of additional bookkeeping from $\mathcal{F}_{\text{CGKA}}$, such as keeping track of two types of history-graph nodes, one for proposals and one for commits (see [9, 28]). Most protocols based on ITK, such as SAIK, can be easily adapted to the propose-commit syntax and benefit from the efficiency gain. The change is minimal and security proofs are clearly not affected.

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

5 THE SAIK PROTOCOL

SAIK inherits most of its mechanisms from ITK, the CGKA of MLS. We briefly recall ITK in Sec. 5.1. Readers familiar with ITK can jump directly to Sec. 5.2 which gives intuition how SAIK improves on ITK. The detailed description of SAIK is in the full version [8].

5.1 Intuition for the ITK Protocol

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

ITK maintains the following *tree invariant*: *Each member knows the secret keys of the nodes on the path from their leaf to the root, and only those, as well as all public keys in* τ. This allows to efficiently encrypt messages to subgroups: If a node v is not blank, then a message m can be encrypted to all members in the subtree of a node v by encrypting it under v's public key. If v is blank, then the same can be achieved by encrypting m under each key in v's *resolution*, i.e., the minimal set of non-blank nodes covering all leaves in v's subtree (Note that leaves are never blank, so there is always a resolution covering all leaves).

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

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

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

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

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

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

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

5.2 Intuition for the SAIK Protocol

mmPKE.. In ITK, a member performing an update generates a sequence of path secrets $s_1, \ldots, s_n$ and encrypts each s_i to each public key from a set of recipient public keys S_i using regular encryption. In contrast, SAIK redraws its internal abstraction boundaries viewing the sequence of encryptions as a single call to mmPKE. This allows it to use the DDH-based mmPKE construction of [32]. Compared to ITK, this cuts the computational complexity of encrypting $\vec{m}$ and the ciphertext size in half (asymptotically as n grows).

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

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

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

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

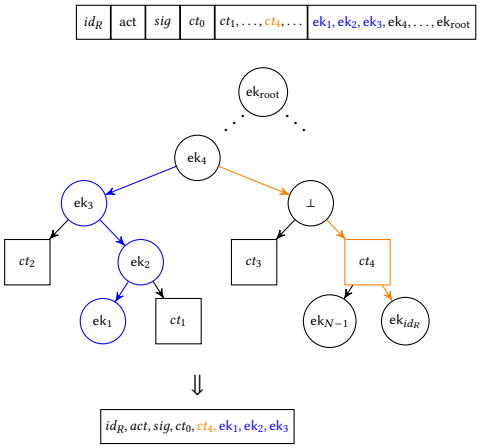

Figure 1: Server extraction algorithm. Lowest common ancestor (LCA) pof id_R and id_S is ek_4, so all blue public keys are included in id_R's packet. Since the sibling of ek_3 is empty, there corresponding path secret of ek_4 is encrypted to its resolution, resulting in the two ciphertext.

Comparison with techniques of [28]. The work [28] introduces a technique for efficient packet authentication which is quite similar to the technique used by SAIK. In particular, their CGKA uses a committing mPKE, cmPKE. A cmPKE differs from mPKE in that encryption outputs a tag T which is a cryptographic commitment to the plaintext and is delivered to each receiver. Since in [28] every recipient of a commit gets the same message, authenticating T is sufficient for CGKA authentication. We highlight a couple of

differences between that technique and ours: First, it is not clear how to use cmPKE in a tree-based CGKA, where a commit executes multiple instances of CmPKE, and hence we end up with multiple tags T, each delivered to a different subset of the group. Second, using the hash of the encrypted message as T does not result in an IND-CCA secure CmPKE, since a hash allows to easily tell which of two messages is encrypted. Therefore, the construction of [28] uses key-committing encryption to both hide and bind the message.

To summarize, CmPKE introduced by [28] is very useful for the CGKA type they consider and may well find more use-cases beyond CGKA. On the other hand, SAIK's solution fits all types o CGKA, does not require additional properties to prove CGKA security and is more direct. Albeit, it is very CGKA-specific.

6 SECURITY OF SAIK

To define the security we prove for SAIK we fix the two safety predicates **confidential** and **authentic** used by $\mathcal{F}_{\text{CGKA}}$. We next give the intuition; see the full version [8] for the pseudocode. We define two versions of the predicates: a stronger and a weaker one. For better exposition, the stronger version is not achieved by SAIK as presented in this work. But at the cost of added complexity SAIK can easily be extended to achieve it, as described Sec. 8.1.

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

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

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

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

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

Security. For the mmPKE scheme we assume a security property called mmOW-RCCA, defined in the full version [8]. The notion is strictly weaker than mmIND-CCA; in [8] we prove the implication. Formally, the AKS is modeled as the functionality $\mathcal{F}_{\text{AKS}}$. SAIK works in the $\mathcal{F}_{\text{AKS}}$-hybrids model, i.e., $\mathcal{F}_{\text{AKS}}$ is available in the real world and emulated by the simulator in the ideal world. The formal proof of Theorem 6.1 and the definition of $\mathcal{F}_{\text{AKS}}$ are given in the full version [8].

Theorem 6.1. *Let $\mathcal{F}_{\text{CGKA}}$ be the CGKA functionality with predicates* **confidential** *and* **authentic** *defined in as in the text (formally in [8]). Let* SAIK *be instantiated with an mmPKE* mmPKE*, a signature scheme* Sig *and* MAC*, and with the* HKDF *functions modelled as a random oracle* Hash*. Let $\mathcal{A}$ be any environment. Denote the output of $\mathcal{A}$ from the real execution with* SAIK *(and, formally, the hybrid functionality $\mathcal{F}_{\text{AKS}}$ from [8]) as $\text{REAL}_{\text{SAIK},\mathcal{F}_{\text{AKS}}}(\mathcal{A})$ and the output of $\mathcal{A}$ from the ideal execution with $\mathcal{F}_{\text{CGKA}}$ and a simulator $\mathcal{S}$ as $\text{IDEAL}_{\mathcal{F}_{\text{CGKA}},\mathcal{S}}(\mathcal{A})$. There exists a simulator $\mathcal{S}$ and adversaries $\mathcal{B}_1$ to $\mathcal{B}_4$ such that*

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

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

7 EVALUATION

We compare the communication complexity or, informally, the "bandwidth" of SAIK, ITK and CmPKE from [28]. For the sake of this comparison (and to simplify the description), one can think of

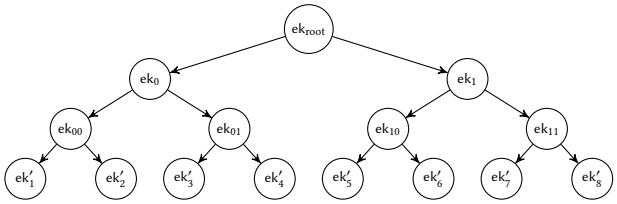

Figure 2: A ratchet tree for SAIK or ITK without blanks or unmerged leaves.

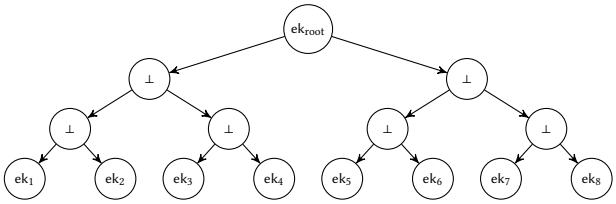

Figure 3: A ratchet tree for SAIK or ITK with all nodes blank.

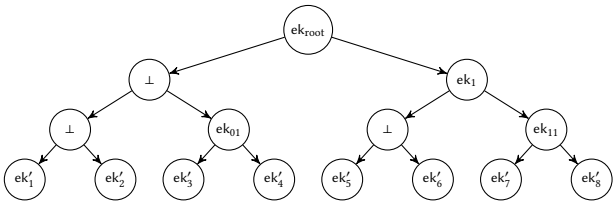

Figure 4: A tree with some blank and non-blank nodes. Here, sender bandwidth depends on the position in the tree. For example, a packet by the leftmost leaf would contain 5 ciphertexts, while the rightmost leaf would require 6.

CmPKE as a protocol similar to SAIK but where the ratchet tree is an N-ary tree of height 1, where N is the number of group members. This means that CmPKE only needs single-message multi-recipient PKE, mPKE (which is a special case of mmPKE). To make a fair comparison, we instantiate CmPKE with the same DH-based mPKE as SAIK instead of the less efficient but post-quantum secure mPKE given in [28].

Methodology. We compare the communication complexity of *a single group modification* with respect to three metrics:

- *sender bandwidth* – the size of the packet uploaded to the server,
- *maximum receiver bandwidth* – the maximum size of a (personalized) packet downloaded by a single receiver, and
- *total bandwidth* – the sum of the sizes of the uploaded packet and all downloaded packets.

The sender and maximum-receiver bandwidths give an idea about the resources a single client needs to invest to perform the group modification. In contrast, the total bandwidth gives the idea about the resources used by the server (or, equivalently, all clients together). However, it makes no assertions about the distribution of this bandwidth, i.e. some clients might use a significantly larger portion of the total bandwidth than others. (We note that the total bandwidth was the (only) metric used in [28].)

There is one caveat when calculating the bandwidth requirements for SAIK (and ITK) due to the underlying tree structure: The bandwidth can vary quite significantly depending on the "tree topology", which is in turn determined by the execution history. Roughly, the reason is that add and remove operations may destroy the good properties of the tree (by "blanking" nodes), increasing the number of public keys to which some message must be encrypted. In the best case, called the *tree-best-case*, there are only $\log(N)$ public keys (this happens when the ratchet has no blank nodes or unmerged leaves, as depicted in Fig. 2). However, in the worst case, called the *tree-worst-case*, there can be N public keys (this happens e.g. when all non-leaf nodes are blank; see Fig. 3). In general, the number of public keys can be anything in between; see Fig. 4.

Therefore, we compare each bandwidth in the tree-best-case and the tree-worst-case. Note that any other case results in a bandwidth between these cases. We remark that comparing the average over all histories of group operations would not be meaningful, since the probability of a given execution depends on user and administrator behavior, general application policies and runtime conditions, etc. It is an important topic of future research to better understand which kinds of policies governing when and which parties initiate CGKA operations lead to more bandwidth efficient executions for realistic deployments. However, it is outside the scope of this work. We note that SAIK is *very* flexible as to kinds of policies that are possible and the types of data that can be leveraged to guide executions towards efficient behavior. Thus, we conjecture that in practice a well designed implementation (of both server and client) will be able to ensure that under relatively mild real-time conditions the vast majority of executions will spend the overwhelming majority of their time tending towards the tree-best-case scenario in bandwidth.

Results. We estimated the bandwidth for all protocols using the formulas in Figs. 5 and 7 with bit lengths indicated in Fig. 8. This is further visualized in Fig. 6 for growing group sizes N. We highlight some interesting observations.

In terms of the sender bandwidth, SAIK is always at least as good as the other protocols. For example, in a group of 10K parties, SAIK sender's require between 83% and 55% of the bandwidth of ITK (due to the smaller mmPKE ciphertexts compared to PKE). SAIK's tree-worst-case sender bandwidth is the same as CmPKE but its tree-best-case bandwidth can be as small as 0.52% by using only 4KB in stead of CmPKE's 783KB. (Recall that, unlike in CmPKE, in SAIK sender bandwidth varies depending on the history of preceding operations.)

In terms of the maximum receiver bandwidth, ITK is much worse than SAIK and CmPKE. For example, SAIK receivers (at most) need between 62% (tree-best-case execution) to about .2% (tree-worst-case execution) of ITK's. On the other hand, CmPKE is the best for receivers. SAIK requires up to 126% of CmPKE's bandwidth, i.e. an increase from $\sim$ 2.4KB to $\sim$ 3.02KB for 10K parties.

Finally, the total bandwidth is by far the smallest for CmPKE and by far the largest for ITK. For instance, for 10K parties, SAIK requires $\sim$ 1.3 times more total bandwidth, while ITK requires anywhere from $\sim$ 2 times (tree-best-case) to $\sim$ 50 times (tree-worse-case) more.

		ITK	SAIK	CmPKE
Sender	tree-best-case	$\log(N) \cdot (\mathsf{Pk} + \mathsf{Ctx})$	$\log(N) \cdot \mathsf{Pk} + \mathsf{mCtx}(\log(N))$	$\mathsf{Pk} + \mathsf{mCtx}(N)$
	tree-worst-case	$\log(N) \cdot \mathsf{Pk} + N \cdot \mathsf{Ctx}$	$\log(N) \cdot \mathsf{Pk} + \mathsf{mCtx}(N)$	$\mathsf{Pk} + \mathsf{mCtx}(N)$
Maximum receiver	tree-best-case	$\log(N) \cdot (\mathsf{Pk} + \mathsf{Ctx})$	$\log(N) \cdot \mathsf{Pk} + \mathsf{Ctx}$	$\mathsf{Pk} + \mathsf{Ctx}$
	tree-worst-case	$\log(N) \cdot \mathsf{Pk} + N \cdot \mathsf{Ctx}$	$\log(N) \cdot \mathsf{Pk} + \mathsf{Ctx}$	$\mathsf{Pk} + \mathsf{Ctx}$
Total	tree-best-case	$N \log(N) \cdot (\mathsf{Pk} + \mathsf{Ctx})$	$N \log(N) \cdot \mathsf{Pk} + N \cdot \mathsf{Ctx} + \mathsf{mCtx}(\log(N))$	$N \cdot (\mathsf{Pk} + \mathsf{Ctx}) + \mathsf{mCtx}(N)$
	tree-worst-case	$N(\log(N) \cdot \mathsf{Pk} + N \cdot \mathsf{Ctx})$	$N \log(N) \cdot \mathsf{Pk} + N \cdot \mathsf{Ctx} + \mathsf{mCtx}(N)$	$N \cdot (\mathsf{Pk} + \mathsf{Ctx}) + \mathsf{mCtx}(N)$

Figure 5: Sender, receiver and total bandwidth for a group of size N expressed as the number of ciphertexts and public keys included in the packet (apart from this, packets include only a constant-size header). Pk denotes the size of a public key (the same for PKE and mmPKE). mCtx(X) denotes the size of an mmPKE multi-recipient ciphertext with overall number of receivers X. Note that for the DH-based construction X fully determines the size (i.e., it is not affected by who gets which message). Ctx denotes the size of a PKE ciphertext, equal to the size of an individual ciphertext in the DH-based construction.

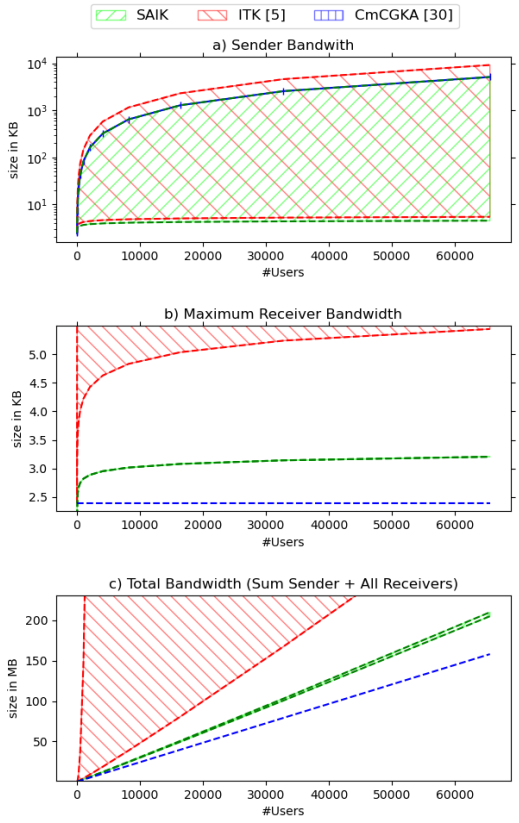

Figure 6: Bandwidth comparison of SAIK, ITK and CmPKE (instantiated with 256-bit security). Lower lines denote the tree-best-case execution history, while upper lines denote the tree-worst-case. All other possible cases are marked as the regions between the lines. Plot (a) shows the sender bandwidth on a *log scale* and plot (b) shows average individual receiver bandwidth on a *linear scale*. Plot (c) shows the total bandwidth, i.e. the sum of sender bandwidth and n times the receiver bandwidth. Note that in the first plot, the lines for tree-worst-case SAIK and all-case CmPKE coincide.

		ITK	SAIK	CmPKE
Sender	best case	$3\log(N)$	$2\log(N)$	N
	worst case	$2N$	N	N
(Maximum) receiver	best case	$3\log(N)$	$\log(N)$	3
	worst case	$2N$	$\log(N)$	3
Total	best case	$3N\log(N)$	$N\log(N)$	$4N$
	worst case	$2N^2$	$N\log(N)$	$4N$

Figure 7: Sender, receiver and total bandwidth for a group of size N expressed as the (approximate) number of group elements. Best/Worst case refers to the state of the tree, while we always consider the average receiver bandwidth over all receivers.

	Bitsize
Group element	512
Hash	512
Signature	1024
Header	17784
Pk	512
Ctx	1152
mCtx(N)	$512 + N \cdot 640$

Figure 8: Bitsizes used to generate Fig. 6. The header consists of the sender's id, the epoch id and some authenticated data required by the protocol. The individual ciphertexts consist of a group element and an AEAD encryption, while the mPKE ciphertext all share the same group element. The header contains signatures, tags, epoch and sender identifier as well as a key package. The latter makes up the bulk of the header, as it contains credentials, more public keys and some application data. Our estimation is based on MLS.

In summary, the results show that SAIK achieves the lowest bandwidth required from a single client (i.e. $O(\log(N))$ for SAIK vs. $O(N)$ for CmPKE), while CmPKE has the lowest total bandwidth (i.e. $O(N \log(N))$ for SAIK vs. $O(N)$ for CmPKE). (Hence, both protocols meet their design goals.) For instance, for 10K parties, CmPKE requires a client to upload 783KB of data, while in scenarios close to the tree-best-case (which we believe to occur most of the time), the sender or receiver bandwidth of SAIK is roughly 4KB. On the other hand, the total bandwidth is roughly 25MB for CmPKE and 30MB for SAIK.

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

8 EXTENSIONS

In this section we describe extensions of SAIK which we did not include for simplicity.

8.1 Better Security Predicates

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

Roughly, SAIK achieves the worse security predicates because of the following attack: Say id_s, the only corrupted party, creates a new epoch E adding a new member id. According to SAIK, in this case id_s fetches from the Authenticated Key Service, AKS, (a type of PKI setup) a public key ek for mmPKE and a verification key spk for Sig, both registered earlier by id. In epochs after E, parties use ek to encrypt messages to id (even before id actually joins) and spk to verify messages from id. Now the adversary $\mathcal{A}$ can create a fake epoch E' adding id with the same ek and spk. Then, id joins E' and is corrupted, leaking dk and ssk. This allows $\mathcal{A}$ to compute the group key in E and inject messages to parties in E. However, the expectation is that this is not possible, since no party is corrupted in E (and id_s healed). The better security predicates (formally, the predicates in the full version [8]) achieve just this: security in an honest epoch E does not depend on whether some member joins a fake group in E'.

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

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

8.2 Primitives with Imperfect Correctness

While the proofs of SAIK security assume primitives with perfect correctness, they can be easily modified to work with imperfect correctness. While most classically secure primitives have perfect correctness, many post-quantum constructions (e.g. from lattices) only have statistical correctness. So this extension can be seen as a preparation for when SAIK has to be adapted to post-quantum security.

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

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

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

ACKNOWLEDGMENTS

Dominik Hartmann was supported by the BMBF iBlockchain project. Eike Kiltz was supported by the BMBF iBlockchain project, the Deutsche Forschungsgemeinschaft (DFG, German research Foundation) as part of the Excellence Strategy of the German Federal and State Governments – EXC 2092 CASA - 390781972, and by the European Union (ERC AdG REWORC - 101054911).

REFERENCES

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

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

[3] Joël Alwen, Margarita Capretto, Miguel Cueto, Chethan Kamath, Karen Klein, Ilia Markov, Guillermo Pascual-Perez, Krzysztof Pietrzak, Michael Walter, and Michelle Yeo. 2021. Keep the Dirt: Tainted TreeKEM, an Efficient and Provably Secure Continuous Group Key Agreement Protocol. 42nd IEEE Symposium on Security and Privacy. (2021). Full Version: https://ia.cr/2019/1489.

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

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

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

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

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

[8] Joël Alwen, Dominik Hartmann, Eike Kiltz, and Marta Mularczyk. 2021. Server-Aided Continuous Group Key Agreement. Cryptology ePrint Archive, Report 2021/1456. (2021). https://eprint.iacr.org/2021/1456.

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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

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